ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ

ВОЛОГОДСКИЙ ГОСУДАРСТВЕННЫЙ
ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

С.Ю. Ржеуцкая

БАЗЫ ДАННЫХ.
ЯЗЫК SQL

Утверждено редакционно-издательским советом университета в качестве учебного пособия

Вологда
2010

УДК 681.3.06
ББК 32.973.233–018.2
Р 48

Рецензенты:

Зейфман А.И., д-р физ.-мат. наук, профессор, декан факультета прикладной математики и компьютерных технологий ВГПУ

Артюгин М.Н., канд. техн. наук, руководитель отдела программных разработок ООО R-Style Softlab Северо-Запад

Ржеуцкая С.Ю.

Р 48 Базы данных. Язык SQL: учеб. пособие / С.Ю. Ржеуцкая. - Вологда: ВоГТУ, 2010. – 159 с.

Учебное пособие предназначено для поддержки лекционного курса по дисциплинам «Базы данных» и «Программирование баз данных». Оно содержит определение основных понятий, формальное описание реляционной модели данных, теоретические и практические аспекты проектирования структуры базы данных, сведения по синтаксису и семантике языка SQL и логике разработки SQL-запросов, вопросы администрирования баз данных.

УДК 681.3.06
ББК 32.973.233–018.2

Оглавление

Введение

Технологии баз данных являются основой, на которой держатся многие компьютерные науки, поэтому уверенные знания в данной области могут служить залогом успешного освоения дисциплин профессионального цикла.

Содержание пособия полностью соответствует стандартной программе курса «Базы данных» ГОС ВПО, используемая терминология согласуется с известными, много раз переиздававшимися фундаментальными работами по базам данных. Основная цель автора пособия - изложить лаконично и систематизированно материал, почерпнутый из большого количества источников, дополнить его собственными примерами и сведениями о современном состоянии некоторых проблем.

Пособие содержит пять глав. В первой определяются основные понятия и дается неформальное введение в системы баз данных. Вторая глава содержит лаконичное, но достаточно строгое описание реляционной модели данных, третья глава посвящена вопросам проектирования реляционных структур. Четвертая глава, самая объемная, содержит систематическое описание языка SQL и одного из его процедурных расширений. Данный материал невозможно изложить чисто абстрактно, поэтому пособие содержит большое количество примеров, которые помогут понять логику разработки запросов и программного кода, хранимого в базе данных. Все примеры ориентированы на одну из самых распространенных коммерческих СУБД Oracle. Наконец, пятая глава посвящена вопросам грамотного использования всех возможностей современных СУБД, здесь тоже в качества примера СУБД используется Oracle.

В качестве дополнения к пособию следует использовать методические указания к лабораторным работам и курсовому проектированию, электронный УМК по базам данных и дистанционный практикум по языку SQL на основе автоматизированной проверяющей системы.

Пособие предназначено для студентов специальностей 220201 – «Управление и информатика в технических системах» и 230105 — «Программное обеспечение вычислительной техники и автоматизированных систем», но может использоваться студентами любых специальностей, которые изучают предметы «Базы данных», «Программирование баз данных», «Базы данных и экспертные системы» и другие родственные дисциплины.

1. Основные понятия

Цель изучения данной главы – овладеть терминологией и понятийным аппаратом дисциплины для эффективного изучения остальных глав пособия.

После изучения главы вы будете:

- иметь общие представления о базах данных, СУБД и информационных системах;
- знать основные архитектуры информационных систем, уметь обосновать выбор архитектуры для решения конкретной задачи
- знать основные модели данных, иметь неформальные представления о реляционной модели
- знать принципы построения информационных систем
- иметь представления об основных СУБД, имеющихся на рынке ПО, уметь грамотно выбрать СУБД в процессе разработки информационной системы

1.1. Терминология, базовые принципы

1.1.1. Понятие базы данных, СУБД и информационной системы

База данных – совокупность структурированных, взаимосвязанных, динамически обновляемых данных определенной предметной области. *Предметная область* – часть реального мира (например, промышленное предприятие, учебное заведение, организация, работающая в сфере услуг и т.д.), подлежащая изучению с целью ее автоматизации. Если предметная область уже автоматизирована, хотя бы частично, но требуется провести мероприятия по реорганизации существующей автоматизированной системы, иногда применяют термин *проблемная область.*

Любая предметная область несет в себе огромное количество *информации,* часть которой, полезную с точки зрения персонала, можно четко выделить, структурировать и сохранить на электронном носителе с целью последующего эффективного поиска и обработки. Структурированная информация предметной области, хранимая на электронном носителе, представляет собой *данные.*

Некоторую часть информации предметной области можно сформулировать в виде *бизнес-правил* — формальных правил, которые учитываются при определении связей между элементами данных. Так формируется база данных, которую можно считать информационной моделью предметной области. Схематически это показано на рис.1.1.

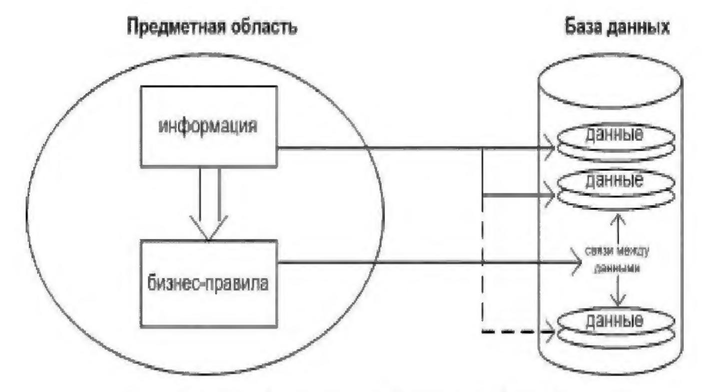


Рис.1.1. Предметная область и база данных

По степени структурированности информации различают *документо-ориентированные* и *фактографические* базы данных. Документо-ориентированные базы содержат слабоструктурированные данные, обычно представленные в виде текстовых документов различных форматов. Фактографические базы данных содержат четко структурированную совокупность данных, основанную на известных в программировании структурах данных.

Способ организации данных и связей в фактографической базе данных называют *моделью данных*. Более строгое определение модели данных содержится в разделе 1.3.

База данных (БД) вместе с поддерживающим ее программным обеспечением (ПО) образует *информационную систему* (ИС). Коротко это можно записать в виде простой формулы БД + ПО= ИС.

Некоторые авторы понимают информационную систему более широко, включая в это понятие еще технические средства и обслуживающий персонал. Иногда встречается и более узкая трактовка информационной системы как совокупность данных и набор программных средств для решения конкретной прикладной задачи, например, задачи бухгалтерского или складского учета.

Классификация ИС

По назначению можно выделить несколько классов ИС:

- ИПС – информационно-поисковые системы. Служат для эффективного поиска информации (примером являются поисковые серверы Интернет);
- УИС (ЭИС) — управляющие (экономические) информационные системы. Такие системы предназначены для автоматизации отдельных функций управления каким-либо экономическим объектом (подразделением, предприятием, корпорацией), поэтому являются важнейшей

частью АСУП (автоматизированной системы управления предприятием). УИС, как правило, содержат подсистему учета данных, отражающих все основные факты деятельности предприятия, и подсистему анализа накопленных данных, которая позволяет руководству предприятия принять грамотное управленческое решение. Такие системы называются *системами поддержки принятия решений* (СППР).

- ЭС — экспертные системы. Способны на самостоятельное принятие решений, т к. имеют в своем составе *базу знаний*, позволяющую получать новые знания на основе уже имеющихся.

По предметной области ИС можно разделить на системы, используемые в производстве, образовании, здравоохранении, науке, военном деле, социальной сфере, торговле и других отраслях.

Состав ИС, персонал, взаимодействующий с системой

Программное обеспечение (ПО) для поддержки базы данных неоднородно. Обычно все ПО подразделяют на *базовое* и *прикладное* (ПрПО).

Базовое ПО включает операционную систему (ОС), которая непосредственно осуществляет доступ к данным на диске, а также специальный комплекс дополнительных программных средств, которые получили название *системы управления базами данных*. Можно рассматривать СУБД как некоторую надстройку над ОС, которая значительно расширяет стандартные возможности ОС по управлению данными.

ПрПО включает программы (приложения), специфичные для конкретной предметной области, которые решают все прикладные задачи, необходимые пользователям системы. Все прикладные программы *взаимодействуют с базой данных только через СУБД*. Взаимодействие программных компонентов информационной системы, а также круга лиц, взаимодействующих с системой, показано на рис. 1.2.

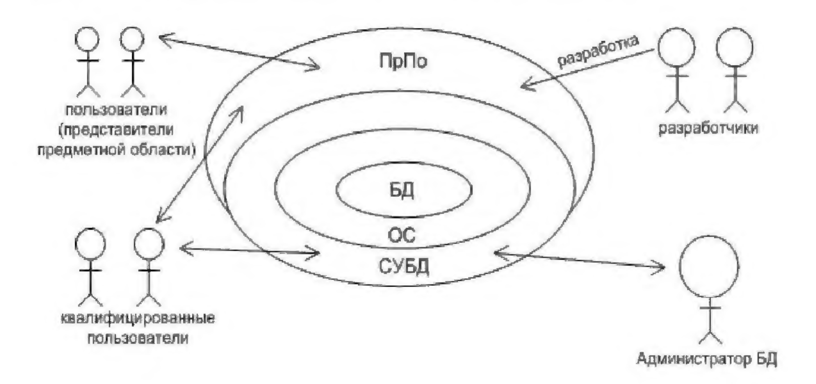


Рис. 1.2. Состав ИС, персонал, взаимодействующий с ИС

Здесь под обычными пользователями понимаются специалисты предметной области, которые используют ИС для автоматизации определенной части своей деятельности (иногда их называют *конечными пользователями*). Они взаимодействуют с БД только через ПрПО. При установке нового ПрПО пользователи проходят курс обучения по его правильному использованию.

Выделим (чисто условно) часть пользователей в группу, которую назовем «продвинутыми» (квалифицированными) пользователями. Представители этой группы имеют некоторое образование в области компьютерных технологий и способны обращаться напрямую к функциям СУБД, если им предоставлены такие права. Для этой цели в СУБД есть различные средства взаимодействия с пользователями, основным из которых является стандартизированный язык запросов к базе данных SQL (подробное описание с примерами содержится в главе 4). Однако основную часть своей работы любые пользователи предпочитают выполнять с удобством, которое им предоставляет ПрПО.

Для успешной работы пользователей системы группа разработчиков подготовила ПрПО и осуществляет его сопровождение. Представители этой группы имеют основное образование в области программирования и компьютерных технологий, они имеют знания в области доступа к базам данных и навыки работы с инструментальными средствами разработки приложений баз данных. Отдельно выделим специалистов в области проектирования структур баз данных. Вопросы проектирования БД подробно обсуждаются в главе 3.

Наконец, немаловажную роль в обеспечении бесперебойного функционирования системы играет *администратор базы данных* (АБД, это может быть один человек или группа лиц). АБД несет ответственность за безопасность и целостность всех данных и осуществляет такие функции, как разграничение доступа пользователей и аудит их действий, регулярное резервное копирование данных и восстановление БД в случае сбоев, обеспечение приемлемой производительности системы, целостности данных и т.д. Этим вопросам посвящена глава 5.

1.1.2. База данных и СУБД

Современные базы данных самодостаточны и относительно независимы от прикладного ПО (на рис. 1.2. видно, что некоторые пользователи работают с базой данных непосредственно через СУБД, минуя слой ПрПО). Такая возможность достигается за счет того, что в современной базе данных хранятся не только сами данные, но и их описание (метаданные, т. е. данные над данными), а также некоторый программный код для обработки данных (рис.1.3).



Рис. 1.3. Состав БД

Часть БД, в которой хранятся данные, получила название *словаря данных* (СД). Словарь данных в том или ином виде присутствует в любой базе данных, независимо от используемой модели данных. Для каждого элемента данных в СД хранится его уникальное имя, тип, размер и некоторые другие свойства. Интересно, что в реляционных базах данных все элементы словаря данных хранятся в таблицах, также как и данные, а для манипулирования элементами словаря используются те же самые реляционные операции, что и для данных.

Дополнительной возможностью, которая поддерживается большинством ведущих производителей СУБД, является хранение программного кода для обработки данных непосредственно в базе данных вместе с данными и метаданными. Более подробная информация о составе базы данных и хранимом програмном коде содержится в главе 4.

СУБД – комплекс программных и языковых средств для создания, ведения и коллективного использования базы данных. В таблице 1.1 приведен список основных функций СУБД, а также языковые и программные средства СУБД, необходимые для реализации каждой функции.

Таблица 1.1

Функции СУБД и средства для их реализации

Функции СУБД:	Языковые средства	Программные средства
1) создание БД и модификация метаданных	Язык DDL (data definition language) в переводе ЯОД (язык определения данных)	Процессор DDL
2) заполнение БД и обновление данных 3) Извлечение данных (выборка)	Язык DML (data manipulation language) в переводе ЯМД (язык манипулирования данными)	Оптимизатор запросов (Query Optimizer)— разработка оптимального плана исполнения запроса пользователя, процессор базы данных (DB Engine)— исполнение запроса по плану
4) обработка данных	Средства разработки хранимого кода - язык высокого уровня, дополненный командами DML или встроенный язык СУБД	Компилятор языка программирования, процессор базы данных

Окончание табл. 1.1

5) обеспечение целостности данных	Правила поддержки целостности (ограничения) в языке DDL, возможность встраивать поддержку целостности в хранимый код	Процессор базы данных, встроенные средства проверки целостности
6) обеспечение безопасности данных (разграничение доступа пользователей и аудит их действий)	Система команд управления доступом к данным	Подсистема безопасности
7) организация коллективного доступа к данным (параллелизм)	Система команд для поддержки транзакций и управления блокировками	Монитор транзакций, подсистема блокировок
8) резервное копирование и восстановление		Утилиты резервного копирования, встроенные средства восстановления БД

1.1.3. Принципы построения информационных систем

Обобщим материал предыдущих разделов, выделив основные принципы. Все принципы очень тесно взаимосвязаны и вытекают одно из другого, поэтому приведенное ниже разделение будем считать условным.

1. *Принцип интегрированности*

Принцип состоит в том, что существует одна единая интегрированная БД для всей предметной области (рис.1.4), которая совместно используется персоналом, при этом одновременно может быть запущено множество приложений (на рисунке П1, П2 и т.д.) с различной функциональностью.

Так, все подразделения одного предприятия исполняют различные функции, но имеют очень тесные информационные связи, поэтому автономная автоматизация каждого подразделения на основе отдельных БД (так называемая «кусочная» автоматизация предприятия) приводит к дублированию данных, избыточным операциям ручного ввода в различных подразделениях, что может привести к нестыковкам данных вследствие ошибок ввода и другим негативным последствиям.

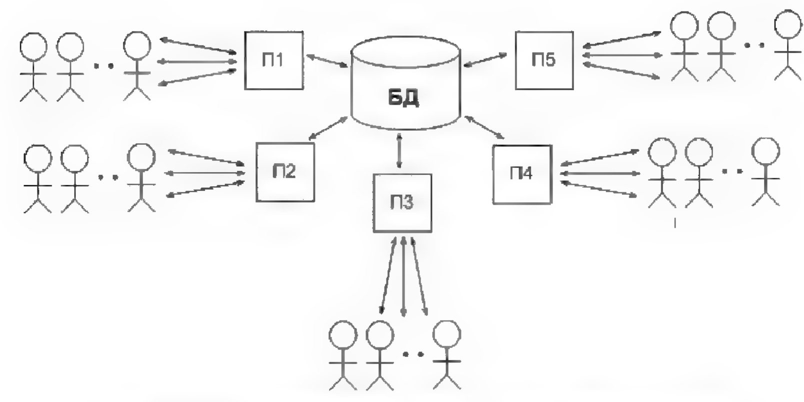


Рис. 1.4. Интегрированная информационная система

В противоположность «кусочной» автоматизации, автоматизация на основе интегрированной информационной системы имеет ряд очень существенных положительных моментов.

- В интегрированной системе может быть достигнута *минимальная избыточность* (отсутствие дублирования) данных. Этот принцип обычно формулируется так: «Каждый факт - в одном месте». В реляционной базе данных некоторая избыточность вносится только для установления связей между таблицами с помощью одинаковых столбцов. Более подробная информация об этом содержится в главе 3 «Проектирование базы данных».
- В интегрированной системе легче добиться непротиворечивости (целостности) данных, т.к. ввиду отсутствия дублирования данных нет и их нестыковок. Имеется возможность контролировать целостность данных встроенными средствами СУБД, более подробная информация об этом содержится в разделе 2.1.
- В интегрированной системе удобнее выполнять поиск и обработку данных, можно выполнять любые виды обработки и анализа данных.
- Для интегрированной ИС проще решается проблема резервного копирования данных и восстановления поврежденных данных, так как эту задачу можно возложить на одного человека (АБД), который будет нести персональную ответственность за сохранность всех корпоративных данных.

Следует отметить, что предприятия, имеющие территориально распределенную структуру, обычно используют так называемые *распределенные базы данных,* в которых отдельные элементы базы данных физически располагаются в различных узлах корпоративной сети. Тем не менее, распределенные базы данных проектируются таким образом, чтобы не нарушать принципа интегрированности.

2. *Принцип независимости прикладного программного обеспечения от способа организации данных.*

Между данными и прикладным программным обеспечением ИС находятся, как минимум, два слоя базового программного обеспечения – операционная система и СУБД, которые берут на себя все низкоуровневые функции управления данными. Поэтому база данных может функционировать и вообще без ПрПО, а одно и то же ПрПО может взаимодействовать с базами данных, имеющими различную физическую организацию.

Различают следующие уровни независимости:

а) логическая независимость – можно вносить некоторые изменения в структуру уже заполненной базы данных без коренной переделки прикладного программного обеспечения, например, можно добавить новые столбцы в уже заполненную таблицу базы данных, при этом все приложения не потеряют работоспособности, однако при удалении столбцов, а тем более таблиц некоторые приложения работать не смогут;

б) физическая независимость – может быть изменен физический формат хранения данных, т.е. переход на новую СУБД или новую версию СУБД, без коренной переделки прикладного программного обеспечения (ПрПО о физическом формате хранения данных вообще ничего «не знает», поскольку работает с данными на логическом уровне).

3. *Принципы масштабируемости и переносимости*

Данные принципы вытекают из принципа независимости данных и ПрПО. Принцип масштабируемости следует рассматривать в трех аспектах:

а) возможность неограниченного наращивания размеров БД;

б) неограниченное увеличение количества пользователей;

в) неограниченное увеличение количества приложений.

В определенный момент времени существующее базовое ПО перестанет удовлетворять требованиям эффективного управления возросшим количеством данных, или не сможет обеспечить приемлемую скорость работы для увеличившегося количества пользователей, поэтому возникнет необходимость перенести данные на новую платформу, что должно быть выполнено без потери информации и коренной переделки ПрПО. Это свойство ИС называется *переносимостью.*

Концепция открытых систем предлагает в качестве механизма воплощения в жизнь перечисленных выше принципов использовать общепризнанные *международные стандарты, регламентирующие все аспекты функционирования информационной системы.* В настоящее время имеющиеся стандарты открытых систем позволяют разворачивать интегрированные *гетерогенные* информационные системы, основанные на использовании разнородного программного обеспечения, обеспечивать их масштабируемость и переносимость.

Далее рассмотрим архитектуры информационных систем подробнее.

1.2. Архитектуры информационных систем

1.2.1. Понятие архитектуры информационной системы

Архитектура – это совокупность существенных решений об организации ИС. Обычно в понятие архитектуры входят решения об основных аппаратных и программных составляющих системы, их функциональном назначении и организации связей между ними.

Выбор архитектуры ИС влияет на следующие характеристики:

1. Производительность ИС – количество работ, выполняемых в ИС за единицу времени.
2. Время реакции системы на запросы пользователя (время отклика системы).
3. Надёжность – способность к безотказному функционированию в течение определенного периода времени.

Локальные ИС, которые располагаются целиком на одном компьютере и предназначены для работы только одного пользователя, сейчас встречаются крайне редко. В дальнейшем речь пойдет о распределенных ИС, которые функционируют в сети и предназначены для многопользовательской (коллективной) работы.

Обычно база данных целиком хранится в одном узле сети, поддерживается одним сервером и доступна для всех пользователей локальной сети, называемых клиентами. Такая база данных называется централизованной. Распределенные базы данных, в которых БД распределена по нескольким узлам сети, обычно используются в организациях, содержащих территориально удаленные подразделения.

Сервер, как правило, — самый мощный и самый надежный компьютер. Он обязательно подключается через источник бесперебойного питания, в нем предусматриваются системы двойного или даже тройного дублирования. В зависимости от распределения функций обработки данных между сервером и клиентами различают две основных архитектуры – «*файл-сервер*» и «*клиент-сервер*». Возможны разновидности этих двух вариантов.

1.2.2. Архитектура «файл-сервер»

Для предприятий малого бизнеса возможна организация информационной системы на базе архитектуры "файл-сервер" с использованием СУБД Access, FoxPro (Visual FoxPro), Paradox и ряда других. Если количество пользователей системы не велико, подобное решение оптимально.

В архитектуре файл-сервер *вся обработка данных выполняется на клиентских компьютерах*, сервер служит в качестве хранилища данных (рис. 1.5).

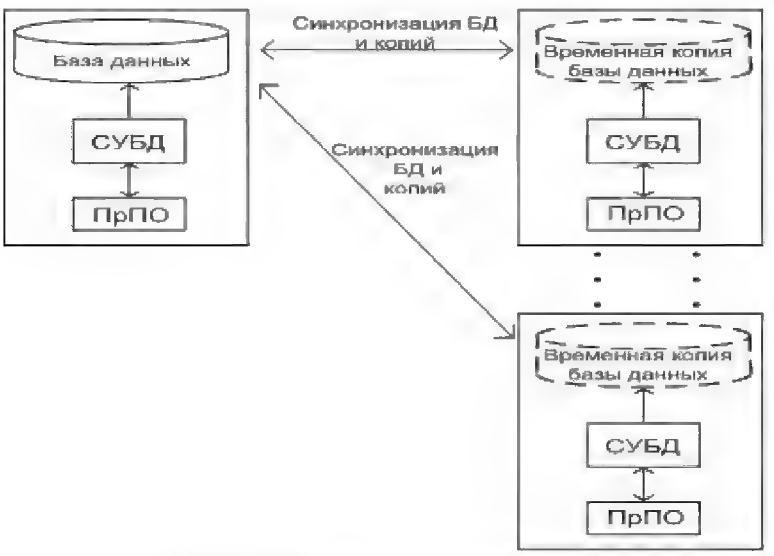


Рис.1.5. Архитектура файл-сервер

Копии базы данных передаются для обработки на клиентские компьютеры, при этом постоянно выполняется синхронизация основной базы данных с ее копиями в случае их обновления.

Недостаток архитектуры файл-сервер – большая нагрузка на сеть и клиентские компьютеры, поскольку на всех клиентских компьютерах должна быть установлена копия СУБД, которая выполняет все необходимые функции по обработке данных, при этом все изменения в копиях обязательно передаются по сети в основную базу данных, существенно повышая сетевой трафик.

Преимущество состоит в том, что не требуется мощный сервер. Такую архитектуру можно реализовать даже в одноранговой сети без выделенного сервера, необходимо только выделить один из компьютеров в качестве хранилища общей базы данных.

Количество пользователей системы в архитектуре файл-сервер обычно не должно превышать 10-15, в противном случае пользователи будут ощущать замедление работы. Данное обстоятельство служит нарушением принципа масштабируемости (раздел 1.1), поэтому по мере роста количества пользователей ИС (допустим, произошло существенное расширение бизнеса) приходится выполнять переход от файл-серверной к

клиент-серверной архитектуре. При разработке файл-серверной системы всегда нужно учитывать возможность такого перехода в будущем.

1.2.3. Архитектура «клиент-сервер»

Применительно к информационным системам архитектура «клиент-сервер» интересна и актуальна главным образом потому, что обеспечивает простое и относительно дешевое решение проблемы коллективного (многопользовательского) доступа к базам данных в локальной или глобальной сети.

Информационная система архитектуры «клиент-сервер» разбивается на две части, которые могут выполняться в разных узлах сети, - клиентскую и серверную части. На серверную часть возлагаются функции хранения и значительной части обработки данных, на клиентскую – функции взаимодействия с пользователем и, частично, обработки данных, полученных с сервера (рис. 1.6).

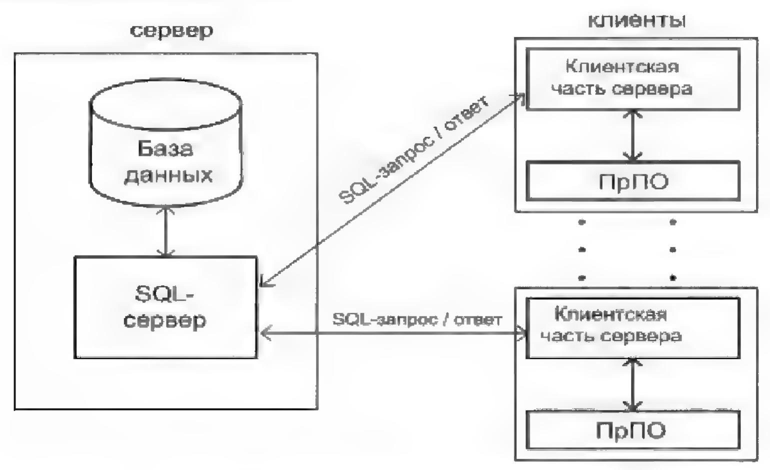


Рис. 1.6. Архитектура «клиент-сервер»

Следует заметить, что обе части системы (серверная и клиентская) могут располагаться и на одном компьютере, такой вариант можно применять в процессе отладки клиент-серверной системы.

Для того, чтобы прикладная программа, выполняющаяся на клиентском компьютере, могла запросить услугу у сервера, требуется некоторый интерфейсный программный слой, поддерживающий взаимодействие сервера с клиентами. Прикладное ПО или конечный пользователь взаимодействуют с клиентской частью системы. Клиентская часть системы

при потребности обращается по сети к серверной части. Интерфейс серверной части определен и фиксирован.

В современных информационных системах таким интерфейсом, как правило, является язык SQL, т.е. сервер принимает от клиентской части SQL-запрос и выполняет необходимые операции в базе данных, после чего возвращает ответ клиенту. По сути, язык SQL представляет собой стандарт интерфейса СУБД в открытых системах (концепция открытых систем затрагивалась в предыдущем разделе).

В системе «клиент-сервер» возможно создание новых клиентских частей уже существующей системы, причем максимальное количество одновременно работающих с общей БД клиентов несравнимо больше, чем в файл-серверной архитектуре, т.е. система клиент-сервер является более масштабируемой. Это объясняется тем, что сетевой трафик в клиент-серверной системе невысок (от клиента передаются только тексты запросов, от сервера – уже отобранные данные, а не вся база данных, как в архитектуре файл-сервер).

Термин «сервер баз данных» обычно используют для обозначения всей СУБД, основанной на архитектуре "клиент-сервер", включая серверную и клиентскую части. Собирательное название SQL-сервер относится ко всем серверам баз данных, основанных на использовании языка SQL.

В настоящее время имеется несколько широко распространенных коммерческих SQL-серверов – Oracle, DB-2, MS SQL Server, Sybase, Informix, Interbase и свободно распространяемые серверы с открытым исходным кодом PostGres (PostgreeSQL), MySQL, FireBird (свободно распространяемый вариант сервера Interbase). Приведенный список далеко не полон.

SQL-серверы обладают преимуществами и недостатками. Очевидное преимущество - стандартность интерфейса. В пределе, хотя на практике это пока не совсем так, клиентские части могли бы работать с любым SQL-сервером вне зависимости от того, кто его произвел. Иными словами, прикладное программное обеспечение на стороне клиента легко настраивается на взаимодействие с любым новым SQL-сервером.

Недостаток – большая нагрузка на сервер, который должен отрабатывать запросы всех клиентов, и малая нагрузка на клиентскую часть. По мере роста количества одновременно работающих пользователей сервер часто становится узким местом всей системы и возникает необходимость его разгрузки. Для этого существуют два пути.

- Если клиентские компьютеры обладают достаточной мощностью, то можно возложить на них больше функций обработки данных, разгрузив сервер.
- В случае применения маломощных клиентских компьютеров (а это более типичная ситуация), применяют многозвенную (многоуровне-

вую) архитектуру «клиент-сервер», выделив промежуточные дополнительные слои программного обеспечения между клиентом и сервером.

1.2.4. Многозвенные архитектуры

Многозвенные (многоуровневые) архитектуры позволяют обеспечить более оптимальную загрузку технических средств, чем классическая двухзвенная архитектура, и обеспечивают возможность плавного масштабирования информационной системы. Наиболее распространенным случаем является трехзвенная архитектура, в которой в качестве промежуточного слоя программного обеспечения между сервером и клиентом используется *сервер приложений* (рис. 1.7). Сервер приложений берет на себя существенную часть обработки данных, позволяя разгрузить и серверную, и клиентскую части.

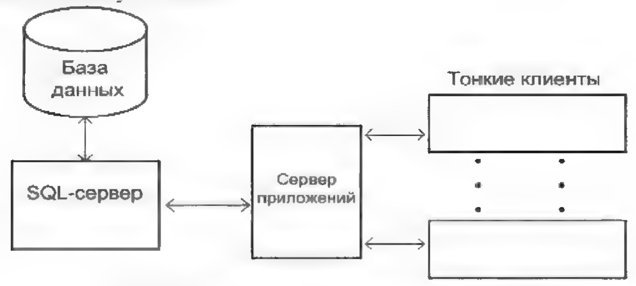


Рис. 1.7. Трехзвенная архитектура

В трехзвенной архитектуре часто используется так называемый *тонкий клиент (thin client)*, который вообще не выполняет никаких функций обработки данных, а только обеспечивает представление данных и взаимодействие с пользователем. В отличие от этого, клиента двухзвенной системы обычно называют *толстым клиентом (fat client)*.

В качестве примера широко используемых ИС трехзвенной архитектуры можно привести любую конфигурацию на базе платформы разработки 1С:Предприятие версии 8 (более ранняя версия 7.7 этой платформы использует либо файл-серверную, либо двухзвенную архитектуру клиент-сервер). В качестве SQL-сервера системы 1С могут использовать MS SQL-сервер, либо свободно распространяемую PostGres, в качестве сервера приложений – сервер 1С.

1.2.5. Информационные системы на основе web-архитектуры

Достижения в области технологий Internet привели к широкому распространению еще одной разновидности клиент-серверной архитектуры, которая получила название web-архитектуры (рис.1.8)

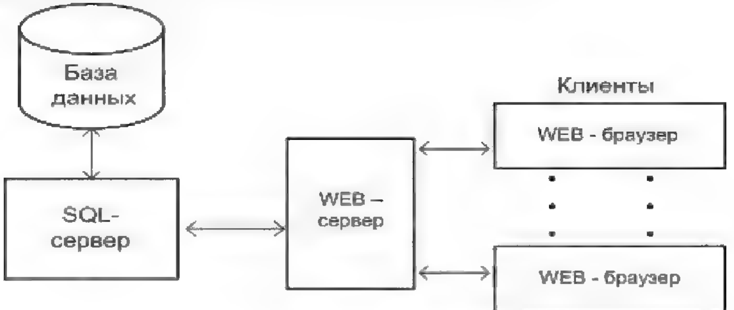


Рис.1.8. Web-архитектура

В web-архитектуре обязательным является наличие еще одного дополнительного компонента – web-сервера, на котором устанавливается прикладное программное обеспечение, обеспечивающее всю необходимую функциональность ИС. В качестве такого сервера может использоваться Internet Information Server (IIS) фирмы Microsoft или свободно распространяемый Web-сервер Apache.

На клиентской стороне требуется только браузер (например, Internet Explorer) для отображения html-страниц, принимаемых со стороны web-сервера, и взаимодействия с пользователем.

В современных ИС между SQL-сервером и Web-сервером часто находится еще одно звено – сервер приложений, который берет на себя большую часть обработки данных и позволяет одновременно разгрузить и SQL-сервер, и Web-сервер. В качестве примера подобной архитектуры можно привести информационные системы на платформе J2EE.

Преимуществом web-архитектуры является удобство администрирования ИС (на клиенте вообще не требуется устанавливать никакого специального программного обеспечения), возможность доступа пользователей к данным как с любого компьютера локальной сети, так и удаленно через Internet.

Широкое применение эта архитектура находит в системах дистанционного образования, системах электронной торговли, социальных сетях и других системах с массовым доступом пользователей, возможно, удаленным.

Недостатком является повышенная угроза безопасности системы, подключенной к глобальной сети Интернет, что усложняет задачу обеспечения безопасности и требует тщательно продуманной системы разграничения доступа пользователей, применения тех или иных методов шифрования данных.

В системах, содержащих конфиденциальные данные или данные, являющиеся коммерческой тайной предприятия, web-архитектура используется редко, при этом обычно устанавливается внутренний Web-сервер, не подключенный к глобальной сети.

1.2.6. Информационные системы, функционирующие в терминальном режиме

Если предприятие имеет мощный сервер и довольно маломощные клиентские компьютеры, оно может развернуть информационную систему в терминальном режиме. При таком способе организации весь программный код как серверной, так и клиентской части исполняется на сервере, а на терминалы передается лишь изображение, которое должно быть отображено на экране монитора (рис. 1.8). Все данные, введенные с терминала, немедленно передаются для обработки на сервер.

Роль терминалов с успехом могут исполнять обычные персональные компьютеры, для этого на каждом из них должна быть запущена специальная программа – эмулятор терминала. В таком режиме может быть развернута и файл-серверная, и клиент-серверная архитектура, в любом случае на сервере должно быть запущено столько копий клиентского программного обеспечения, сколько терминалов подключено к серверу (рис. 1.9).

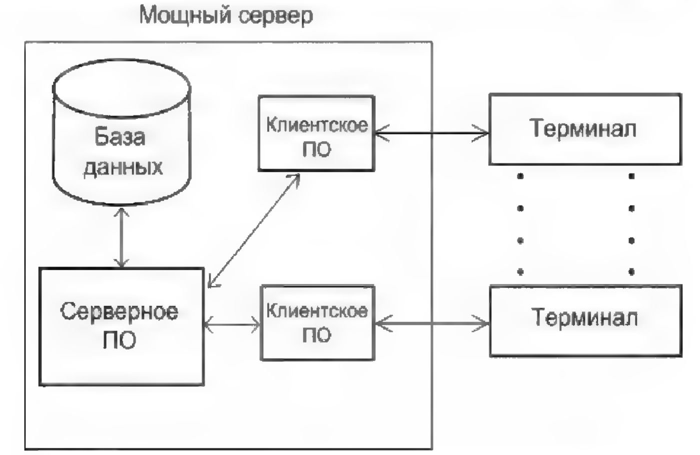


Рис. 1.9. Терминальный вариант развертывания ИС

Преимущества такого варианта налицо – простота и удобство администрирования (все ПО сосредоточено на одном сервере), повышенная безопасность, самые низкие требования к клиентским компьютерам. Однако требования к серверу, на котором сосредоточено функционирование все системы, очень высокие, а выход его из строя даже на короткое время может расцениваться как катастрофа, поскольку вся система становится неработоспособной.

1.3. Модели данных

Модель данных в общем виде можно представить в виде трех составляющих:

- логическая структура данных (в первую очередь, способ организации связей между элементами данных):
- набор допустимых операций по манипулированию данными для принятой логической структуры;
- правила поддержки целостности (непротиворечивости) данных для принятой логической структуры.

Теоретические основы баз данных и языков программирования довольно долго развивались параллельно, поэтому концепция абстрактного типа данных (АТД), которая давно и прочно утвердилась в программировании, в базах данных также довольно долго и устойчиво развивается в терминах модели данных. Тем не менее, можно считать понятия «модель данных» и «абстрактный тип данных» очень близкими по смыслу. Изучая предмет «Базы данных», будем придерживаться его терминологии.

1.3.1. Сравнительная характеристика моделей данных

К основным моделям данных относят следующие:

- иерархическая (на основе деревьев),
- сетевая (на основе многосвязных структур—графов),
- реляционная (на основе таблиц),
- постреляционная (таблицы с возможностью вложения одних таблиц в другие),
- объектно-ориентированная (на основе принципов объектно-ориентированного программирования).

Иерархическую и сетевую модель в настоящее время следует признать интересными лишь в историко-теоретическом плане, поскольку первые СУБД, появившиеся в 60-е - 70-е годы, поддерживали именно эти модели. Безусловно, многие полезные идеи, отработанные в рамках ие-

рархических и сетевых СУБД, находят применение и в современных системах, основанных на реляционной модели и ее развитии в виде постреляционной и объектно-ориентированной модели.

Все приведенные выше модели данных основаны на структурах, хорошо известных в программировании, однако их применение для организации баз данных имеет определенную специфику, поэтому кратко остановимся на особенностях каждой из моделей.

1. Иерархическая модель данных

В СУБД иерархического типа вся информация представлена в виде деревьев, узлами которых являются записи (например, записи о подразделениях предприятия, сотрудниках подразделений и их детях образуют дерево). Корневая запись каждого экземпляра дерева (в нашем случае запись о подразделении) обязательно должна содержать ключ с уникальным значением. Ключи некорневых записей должны иметь уникальное значение только в рамках данного экземпляра дерева. Каждая запись идентифицируется полным сцепленным ключом - совокупностью ключей всех записей, начиная от корневой, по иерархическому пути.

Каждый экземпляр дерева называется *групповым отношением*, корневая запись называется *владельцем* группового отношения, а все дочерние записи – *членами* группового отношения.

Говорят, что иерархическая модель реализует отношение «один-ко-многим» (one-to-many) между исходной и дочерней записью, поскольку каждому экземпляру исходной записи соответствует несколько экземпляров дочерних записей. Такое отношение обозначается как 1:M или 1:N.

Например, в каждом подразделении предприятия может работать несколько («много») сотрудников, но каждый сотрудник принадлежит только одному подразделению.

В иерархических БД поддерживается целостность связей между владельцами и членами группового отношения (никакой потомок не может существовать без своего предка).

Недостатки иерархических БД:

Кроме отношения «один-ко-многим» (1:M), довольно распространенным на практике является отношение «многие-ко-многим» (M:M, many-to-many). Например, в приведенном примере с подразделениями и их сотрудниками бизнес-правила предприятия могут допускать возможность работы сотрудника и в нескольких подразделениях одновременно. Такого сотрудника придется включить в экземпляры деревьев для всех подразделений, в которых он работает. То же самое можно сказать и об отношении «сотрудники - дети», поскольку возможна ситуация, когда оба родителя ребенка работают на одном предприятии и даже в одном подразделении. Таким образом, возникает дублирование информации.

Запросы на поиск информации в иерархической базе данных выполняются по-разному. Поиск, выполняющийся в направлении от корневой к дочерним записям (например, получить сведения обо всех сотрудниках бухгалтерии), выполняется очень эффективно, но поиск в обратном направлении (например, выяснить, в каком подразделении работает родитель Пети Иванова), наоборот, выполняется крайне медленно.

В настоящее время иерархическая модель используется редко, в основном, для отдельных специальных применений. Например, реестр Windows представляет собой иерархическую базу данных.

Широко распространенных коммерческих или свободно распространяемых СУБД, поддерживающих иерархическую модель, в настоящее время нет.

2. Сетевая модель данных

Сетевая модель данных определяется в тех же терминах, что и иерархическая. Она состоит из множества записей, которые могут быть владельцами или членами групповых отношений. Связь между записью-владельцем и записью-членом также имеет вид 1:М.

Основное различие этих моделей состоит в том, что в сетевой модели запись может быть членом *более чем одного* группового отношения. Продолжая пример с подразделениями и сотрудниками, в сетевой модели можно включить запись о сотрудникс в групповыс отношсния для различных подразделений. Таким образом, в сетевой модели данных, в отличие от иерархической, дублирование информации отсутствует.

Согласно этой модели, каждое групповое отношение именуется и проводится различие между его типом и экземпляром. Тип группового отношения задается его именем и определяет свойства, общие для всех экземпляров данного типа. Экземпляр группового отношения представляется записью-владельцем и множеством (возможно пустым) подчиненных записей.

Отсутствие дублирования информации, безусловно, является самым главным достоинством сетевой модели данных. *Недостатком сетевой модели* является ее чрезмерная сложность, что в период распространения сетевых СУБД приводило к большим проблемам в администрировании сетевых баз данных. Например, проблема восстановления поврежденных или утраченных данных решалась ценой огромных усилий. Некоторые запросы в сетевых базах, как и в иерархических, выполнялись очень медленно.

Очевидно, в силу указанных выше недостатков, сетевые СУБД практически прекратили свое существование.

При реализации иерархических и сетевых баз данных был принят принцип, который, как показала практика, имеет серьезные недостатки.

Для связывания записей, принадлежащих одному групповому отношению, в этих моделях использовались указатели, представляющие собой физические адреса данных на диске. Недостатки такого подхода были проанализированы в работах Е.Ф.Кодда, которого по праву считают автором и идеологом реляционной модели данных.

3. Реляционная модель данных

В 1970 г Е.Ф.Кодд опубликовал 2 статьи, в которых ввел реляционную модель данных и реляционные языки обработки данных - реляционную алгебру и реляционное исчисление. В своей работе Кодд продемонстрировал недостатки существующих подходов к связыванию данных с помощью хранения физических адресов данных (указателей). Он показал, что такие базы данных существенно ограничивают число типов манипуляций данными. Более того, они очень чувствительны к изменениям в физическом окружении. Когда в компьютерной системе устанавливался новый накопитель или изменялись адреса хранения данных, требовалось дополнительное преобразование файлов. Если к формату записи в файле добавлялись новые поля, то физические адреса всех записей файла изменялись. То есть такие базы данных не позволяли манипулировать данными так, как это позволяла бы логическая структура. Все эти проблемы преодолела реляционная модель.

В реляционной модели достигается гораздо более высокий уровень абстракции данных, чем в иерархической или сетевой модели. В статье Е.Ф.Кодда утверждается, что "реляционная модель предоставляет средства описания данных на основе только их естественной структуры, т.е. без потребности введения какой-либо дополнительной структуры для целей машинного представления". Другими словами, представление данных не зависит от способа их физической организации.

Основным логическим объектом для хранения данных в реляционной модели является таблица. Для организации связей между данными различных таблиц используются общие столбцы. Например, для примера с подразделениями и сотрудниками понадобятся две таблицы (одну можно назвать Подразделения, другую - Сотрудники), связанные общим столбцом. Таким столбцом может быть, например, Личный_код_сотрудника (или табельный номер сотрудника). Более подробное неформальное введение в реляционную модель содержится в следующем разделе данной лекции.

Эта простая идея связывания таблиц оказалась столь жизнеспособной, что уже на протяжении свыше 30 лет реляционная модель является основной в базах данных. Огромное количество данных уже реально хранится на магнитных носителях в виде таблиц, хорошо проработаны теоретические основы такого способа хранения данных. В силу указанных

обстоятельств новые модели данных вводятся очень осторожно, чтобы не разрушить уже функционирующие информационные системы. Тем не менее, такие модели данных постепенно вводятся и поддерживаются производителями СУБД.

4. Постреляционная модель данных

Постреляционная модель в основе содержит реляционную модель, дополненную возможностью создания вложенных таблиц. Постреляционная модель уже поддерживается некоторыми СУБД, например, Oracle, Postgres и рядом других, которые содержат такие типы данных как массивы и таблицы. Однако реальное использование этих новых типов данных встречается довольно редко, поскольку СУБД пока не гарантируют такую же эффективность работы с вложенными таблицами, как с таблицами, содержащими только атомарные (неделимые) значения.

Если продолжить пример с подразделениями и сотрудниками, то к таблице Сотрудники можно было бы добавить столбец Дети, представляющий собой вложенную таблицу с именами и датами рождения детей сотрудника. Можно было бы добавить и столбец Образование, содержащий названия учебных заведений, которые закончил данный сотрудник, а также даты их окончания и номера дипломов.

5. Объектно-ориентированная модель данных

Объектно-ориентированная модель является очень перспективной в связи с распространением объектно-ориентированного подхода к разработке программных продуктов. На сегодняшний день ее распространение сдерживают два обстоятельства:

- Отсутствие строгой математической модели объектно-ориентированной базы данных. Для реляционной модели такое строгое описание имеется;
- Наличие огромного количества данных в имеющихся реляционных базах данных и существенные затраты на их конвертацию в объектно-ориентированную БД.

В силу этих обстоятельств внедрение объектно-ориентированного подхода в базы данных происходит эволюционно, без разрушения реляционной основы. На сегодняшний день многие СУБД позиционируются как объектно-реляционные. В их основе по-прежнему лежит реляционная модель, но она дополнена возможностью создания пользовательских типов столбцов с поддержкой принципов инкапсуляции и наследования.

1.3.2. Неформальное введение в реляционную модель

Формальное определение реляционной модели основано на теории множеств (математической моделью таблицы является *отношение – relation,* отсюда и произошел термин реляционная база данных). Е.Ф. Кодд определил систему операций над отношениями (реляционную алгебру) и сформулировал основные правила поддержки целостности реляционной базы данных. Им же был предложен и язык для манипулирования реляционной базой данных, который тогда получил название SEQUEL, а впоследствии превратился в язык SQL.

Изучение языка SQL невозможно без знания основ реляционной модели, которую он полностью поддерживает, поэтому в следующих лекциях будет приведено ее формальное описание, основанное на работах Е.Ф.Кодда и К.Дж.Дейта. В качестве введения в реляционную модель далее приведем неформальные, но достаточно строгие определения.

1. Таблицы и связи

Доктор Кодд выделил 12 принципов (правил) реляционных баз данных. Начнем с первого правила, которое позволит понять суть реляционной модели:

● *вся информация логически представлена в виде таблиц.*

Таблица состоит из строк и столбцов (в реляционной теории строке соответствует *кортеж*, а столбцу — *атрибут*, однако в стандарте SQL используются общепринятые термины «строка» и «столбец»).

Поскольку вся информация, хранимая в базе данных, не может быть размещена в одной таблице, возникает вопрос, как организовать связи между таблицами. Согласно первому правилу Кодда, вся информация, в том числе и информация о связях между данными, должна содержаться в самих таблицах. В реляционной модели для этого используются *общие столбцы таблиц.*

Например, пусть в базе данных коммерческой фирмы необходимо хранить ряд сведений о клиентах фирмы:

- порядковый номер (личный код) клиента;
- фамилия, имя, отчество;
- контактные телефоны.

При попытке разместить все эти данные в одной таблице возникает серьезная проблема со столбцом «контактные телефоны». Обычно клиенты оставляют несколько номеров – рабочий, домашний, мобильный телефон и т.д., причем количество номеров может быть различным (допустима даже такая ситуация, когда клиент вообще не предоставляет ни одного

номера). Наилучшим способом для хранения контактных телефонов будет отдельная (*детальная* или *подчиненная*) таблица, связанная с *основной* таблицей данных о клиентах при помощи общего столбца «личный код клиента».

Возможный вариант заполнения таблиц показан на рис. 1.10.

Клиенты

код	фамилия	имя	отчество
1	Иванов	Иван	Иванович
2	Петров	Петр	Петрович

Контактные телефоны

код	телефон	пояснение
1	111111	рабочий
1	222222	домашний
1	333333333	мобильный
2	123456	рабочий

Рис. 1.10. Информация о клиентах с указанием контактных телефонов

В данном примере клиенту с кодом 1 соответствуют три связанных строки в таблице *Контактные телефоны* (у них всех в столбце *код* стоит значение 1), а клиенту с кодом 2 – только одна связанная строка. Получилась очень стройная и логичная структура из двух связанных таблиц, с помощью которой одинаково легко найти как все номера телефонов конкретного клиента, так и определить, какому клиенту принадлежит тот или иной телефон.

Рассмотрим еще один пример организации связи между таблицами в реляционной базе данных. например, товары и поставщики. Пусть необходимо хранить сведения о товарах и их поставщиках. В данном случае связь между товарами и поставщиками несколько сложнее, чем между клиентами и их телефонами. Каждый товар может поставлять, как правило, несколько фирм-поставщиков, но и каждая такая фирма обычно поставляет несколько товаров. В этом случае наилучшим решением будет создание еще одной таблицы, которую называют таблицей-связкой, для хранения информации о связях между товарами и их поставщиками.

Пример заполнения такой базы данных показан на рис. 1.11, здесь таблица-связка названа, как обычно принято, Товары-поставщики и содержит всего два столбца – код товара и код поставщика. В таблицах-связках могут быть и дополнительные, уточняющие, столбцы, например, цена данного товара у данного поставщика и т.д.

Товары

Код товара	наименование
1	скатерть
2	салфетка
3	наволочка

Товары-поставщики

Код товара	Код поставщика
1	1
2	1
1	2
3	1

Поставщики

Код поставщика	название
1	Русский текстиль
2	Льнокомбинат

Рис. 1.11. Информация о товарах и их поставщиках

2. Первичные, альтернативные и внешние ключи

Обобщим материал приведенных примеров. Рассмотрим, например, связь между таблицами *Клиенты* и *Контактные телефоны.* В таблице *Клиенты* столбец «код» является уникальной характеристикой каждого клиента, такой столбец называется *первичным ключом* (Primary Key - PK). Прилагательное «первичный» добавляется в связи с тем, что уникальных столбцов или уникальных сочетаний нескольких столбцов в реальных таблицах может быть несколько. Например, в таблицу *Клиенты* можно было бы добавить столбец серия и номер паспорта (или два столбца – серия и номер отдельно), и это был бы *альтернативный ключ* таблицы (если уникальный ключ состоит из нескольких столбцов, то он называется *составным*). Заметим, что сочетание фамилии, имени и отчества нельзя считать альтернативным ключом ввиду наличия полных однофамильцев. Сочетания двух столбцов «код» и «фамилия» или «код» и «имя», конечно, обладают свойством уникальности, но это явно избыточные сочетания (фамилию или имя можно отбросить без потери уникальности). Таким образом, ключ таблицы должен удовлетворять двум признакам — уникальности и безизбыточности.

Второе правило Кодда гласит — *логический доступ к данным осуществляется по имени таблицы, имени столбца и значению первичного ключа.* Эти три составляющих позволяют однозначно получить любой элемент реляционной базы данных. Например, по таблице *клиенты,* столбцу *фамилия* и значению кода 2 легко получить фамилию *Петров.*

Связь главной и подчиненной таблиц обычно осуществляется с помощью первичного ключа главной таблицы, который помещается (экспортируется) в подчиненную таблицу и становится там *внешним ключом* (Foreign Key - FK). Внешний ключ не обладает свойством уникальности (каждому клиенту может соответствовать несколько номеров телефонов,

каждому поставщику – несколько товаров), обычно он является частью составного первичного ключа или неключевым столбцом.

В примере с клиентами и телефонами подчиненная таблица имеет составной первичный ключ «код» и «номер телефона» - вместе они образуют уникальное и безизбыточное сочетание. Связь между таблицами *Клиенты* и *Контактные телефоны* называют связью *«один ко многим»* (тут прямая аналогия с иерархической и сетевой базами данных).

В примере с товарами и поставщиками, как правило, наименования всех товаров и названия всех фирм-поставщиков сами по себе уникальны. Однако введение дополнительных ключевых столбцов, иначе называемых *суррогатными ключами,* является повсеместной практикой. Допустим, если какая-либо фирма-поставщик изменила название, чтобы отразить это изменение в базе данных, достаточно исправить всего одно значение в таблице *Поставщики.* Изменять суррогатный ключ при этом не нужно, поэтому информация в таблице-связке останется прежней.

3. NULL-значения

NULL-значения — это *неопределенные* или *пустые значения* данных, которые в реляционной теории трактуются как отсутствие информации (правило Кодда №3). Их нельзя рассматривать как нулевые значения числовых полей или пустые строки в текстовых полях. Допустимость пустых значений в том или ином столбце необходимо указывать при определении таблиц.

В отношении ключевых столбцов справедливы следующие правила:

- в первичном и альтернативном ключах NULL-значения не допускаются;
- во внешнем ключе наличие NULL-значений допустимо, однако при определении таблицы их можно запретить.

Существуют определенные правила обработки NULL-значений. Два NULL-значения никогда не равны друг другу.

4. Метаданные. Схема базы данных

Описание структуры базы данных называется *метаданными* (данные о данных). Метаданные должны храниться в базе данных, как и все прочие значения, а не в прикладной программе (правило Кодда №4).

Для наглядного представления структуры базы данных, удобного для пользователей, используется ее графическое изображение, которое называется *схемой.* Изобразим логическую схему базы данных *Сведения о клиентах,* воспользовавшись международным стандартом IDEF1X (рис.1.12).



Рис. 1.12. Логическая схема базы данных Сведения о клиентах

Как видно из схемы, главная и подчиненная таблицы неразрывно связаны друг с другом, и эта связь имеет логический характер, не зависящий от физического способа реализации таблиц.

5. Правила ссылочной целостности

Внешний ключ подчиненной таблицы можно рассматривать как *ссылку* на строку главной таблицы с таким же значением первичного ключа. Отсюда вытекает основное *правило ссылочной целостности* – все значения внешнего ключа должны быть *согласованы* с соответствующими значениями первичного ключа. Применительно к нашему примеру это означает, что в таблице *контактные телефоны* не должно быть кодов клиентов, которых нет в основной таблице *клиенты.*

Ссылочная целостность должна жестко контролироваться при выполнении любых операций с данными таблиц (правило Кодда №12).

Правило Кодда №7 устанавливает 4 таких операции – извлечение, вставка новых строк, удаление строк и изменение (обновление) существующих строк (операции select, insert, delete, update). Операция извлечения не может нарушить целостности, т.к. не изменяет данные. Рассмотрим основные стратегии поддержки ссылочной целостности для остальных операций.

Операция вставки критична только для подчиненной таблицы (можно добавить нового клиента, у которого нет телефона, но нельзя добавить телефон несуществующего клиента), поэтому вставка новых строк в подчиненную таблицу должна проверяться на согласованность значений внешнего ключа.

Удаление строк подчиненной таблицы, наоборот, абсолютно безопасно (можно удалить любой телефон, не нарушив ссылочной целостности). Однако удаление строк главной таблицы при наличии связанных строк в подчиненной таблице непременно приведет к нарушению ссы-

лочной целостности, чего допустить нельзя ни в коем случае. Здесь имеется две основных стратегии удаления:

- запрет удаления таких строк (ограничение удаления - restrict),
- каскадное удаление строки главной таблицы вместе со всеми связанными строками подчиненной таблицы (удаление каскадом - cascade).

В нашем конкретном примере с телефонами разумным решением будет каскадное удаление (при удалении клиента автоматически должны удаляться и все строки с его телефонами). В реальной практике чаще применятся запрет удаления строк при наличии на них хотя бы одной ссылки из других таблиц. Отметим, что ни одна СУБД не позволит удалить всю таблицу целиком, если имеется хотя бы одна другая таблица, которая на нее ссылается.

Изменение (обновление) значений внешнего, а особенно первичного ключа заполненной базы данных обычно не происходит, однако стратегия каскадного обновления некоторыми СУБД поддерживается.

В следующей главе пособия реляционная модель данных будет определена более формально. Наличие формального математического описания считается одним из достоинств реляционной модели.

2. Реляционная модель

Цель изучения данной главы – освоить реляционную модель данных как математическую основу языка SQL.

После изучения главы вы будете:

- знать реляционную терминологию, понимать определения основных реляционных терминов
- знать операции реляционной алгебры, отличия реляционной алгебры от реляционного исчисления, понимать логику разработки запросов в терминах реляционной алгебры и реляционного исчисления
- знать правила поддержки целостности данных при выполнении основных операций манипулирования данными.

В реляционной модели выделяют три части:

- Структурная часть
- Манипуляционная часть
- Целостная часть

Рассмотрим каждую из них подробнее.

2.1. Реляционная модель. Структурная и целостная части

2.1.1. Структурная часть

Реляционная модель данных является математической основой языка SQL, в которой приведено строгое формальное описание всех реляционных объектов, и ее изучение интересно именно с этой точки зрения.

Такие понятия, как таблица, строка, столбец и т.д., являются точными, но недостаточно формальными терминами для строгого математического описания реляционных объектов. При разработке реляционной модели за основу была принята теория множеств, а в качестве математической модели таблицы используется *отношение (relation),* которое определяется в терминах теории множеств. Собственно, и термин «реляционная» произошел от английского термина relation (иногда в литературе используется термин «реляция»). Соответствие между реляционной и общепринятой терминологией приведено ниже.

Соответствие между реляционными и общепринятыми терминами

Общепринятый термин	Реляционный термин
Таблица	Отношение
Строка	Кортеж
Столбец	Атрибут
Множество допустимых значений столбца	Домен, на котором определен атрибут
Количество столбцов	Степень или арность
Количество строк	Кардинальное число

Для каждого реляционного термина приведем его формальное определение.

2.1.2. Атрибуты и домены. Схема отношения

Атрибут определяется как именованный атомарный (неделимый) элемент данных. Множество допустимых значений атрибута называется *доменом,* на котором определен данный атрибут. Каждый атрибут определен на своем домене, при этом допустимы одинаковые домены для различных атрибутов. Сравнивать между собой можно только значения атрибутов, определенных на одинаковых доменах.

Определение домена очень близко к определению типа данных в языках программирования. В самом общем виде домен определяется заданием некоторого базового типа данных и произвольного логического выражения, применяемого к каждому элементу данных. Если вычисление этого логического выражения дает результат "истина", то элемент данных является элементом домена.

В большинстве реляционных СУБД объект «домен» не используется, но для каждого атрибута задается базовый тип и ряд *ограничений (constraints),* которые проверяются для каждого значения атрибута. В последнем стандарте SQL специфицирована команда *create type...* , которая позволяет конструировать произвольные типы данных, что еще точнее соответствует понятию домена, введенного Коддом для реляционных баз данных.

Схема отношения - это именованное множество упорядоченных пар

(имя атрибута : имя домена).

Степень или "арность" схемы отношения - мощность этого множества (количество элементов, входящих в множество). Например, если в схему отношения входит всего два атрибута, то отношение называется бинарным, если три – тернарным.

Схема БД - это набор именованных схем отношений.

2.1.3. Кортежи. Отношение

Кортеж, соответствующий данной схеме отношения, - это множество упорядоченных пар

(имя атрибута : значение атрибута),

которое содержит по одному вхождению каждого имени атрибута, принадлежащего схеме отношения.

Значение атрибута является допустимым значением из домена данного атрибута. Арность кортежа, т.е. число элементов в нем, совпадает с

арностью соответствующей схемы отношения. Таким образом, можно считать кортеж математической моделью одной (любой) строки таблицы.

Отношение - это множество кортежей, соответствующих одной схеме отношения. В определении Дейта [1] схема отношения называется *заголовком* отношения, а множество кортежей - *телом* отношения. Заголовок отношения соответствует заголовку («шапке») таблицы, тело отношения соответствует всей совокупности данных, содержащихся в таблице.

Из определения отношения следуют его основные свойства:

- в отношении не может быть двух одинаковых кортежей (согласно определению множества, все его элементы уникальны),
- кортежи не упорядочены, атрибуты также не упорядочены (это свойство также является неотъемлемым свойством любого множества).

Добавим к этому, что имена всех атрибутов в пределах одного отношения должны быть уникальны.

Реляционная база данных - это набор отношений, имена которых совпадают с именами схем отношений в схеме БД.

2.1.4. Потенциальные ключи. Первичный ключ

Согласно свойствам отношений, каждый кортеж в целом уникален, однако отдельные атрибуты могут содержать и повторяющиеся значения. Тем не менее, в каждом отношении можно выделить хотя бы одну группу (подмножество) атрибутов (возможно, один атрибут), содержащих гарантированно уникальные значения.

Потенциальным ключом отношения (Candidate Key - CK) называют подмножество атрибутов отношения, которое удовлетворяет двум свойствам:

1. Уникальность (не существует двух одинаковых значений);
2. Безизбыточность (никакое подмножество потенциального ключа не является потенциальным ключом).

Различают простые и составные потенциальные ключи (например, серия и номер паспорта – составной потенциальный ключ, а ИНН - простой).

В каждом отношении можно выделить один или несколько потенциальных ключей. Если таких ключей несколько, один из них выбирается в качестве *первичного ключа (Primary Key - PK).*

Первичные ключи используются в качестве общих столбцов для связывания таблиц. Поэтому в качестве первичного ключа обычно выбирают самый короткий, чаще всего, числовой атрибут, для которого гарантируется не только уникальность, но и неизменность значений. На практике часто невозможно выделить такой атрибут, в этом случае используют

дополнительный атрибут, называемый суррогатным ключом, который автоматически заполняется уникальными значениями и никогда не изменяется.

Все потенциальные ключи отношения, которые не являются первичным ключом, называются альтернативными ключами.

Ни в одном из потенциальных ключей NULL-значения недопустимы.

2.1.5. Внешние ключи

Первичные ключи отношений используются в качестве общих атрибутов при связывании отношений. Для этой цели вводятся понятия *родительского (главного) отношения* и *дочернего (подчиненного) отношения.* Первичный ключ родительского отношения, экспортированный в дочернее отношение в качестве связующего атрибута, называется *внешним ключом дочернего отношения.* Посредством внешнего ключа кортежи дочернего отношения ссылаются на соответствующие им кортежи родительского отношения.

Определим данное понятие более формально. Назовем *внешним ключом (foreign key - FK)* такое подмножество атрибутов дочернего отношения, что для любого его непустого значения обязательно найдется равное значение первичного ключа главного отношения.

Обратное утверждение несправедливо, т.е. в родительском отношении могут найтись такие кортежи, на которые не ссылаются никакие кортежи дочернего отношения. Например, в отношении Сотрудники могут быть кортежи, для которых нет ни одного связанного кортежа в отношении Дети_Сотрудников.

Значения внешних ключей не обязаны быть (и обычно не бывают) уникальными в своем отношении (продолжая пример с сотрудниками и детьми – у сотрудников может быть несколько детей, все они содержит одинаковые значения внешнего ключа).

Во внешнем ключе NULL-значения допустимы, в отличие от первичного ключа, однако на практике такая необходимость встречается крайне редко.

2.1.6. Целостная часть реляционной модели

Целостность данных - это механизм поддержания базы данных в непротиворечивом состоянии, соответствующем динамично изменяющейся предметной области.

Угроза нарушения целостности данных возникает при выполнении операций манипулирования данными. Поэтому все СУБД должны контролировать операции вставки (Insert), удаления (Delete) и обновления

(Update) и отказывать в выполнении операции, если в ней проводится попытка нарушить целостность базы данных. Эта проблема решается путем введения специальной системы мер, не позволяющих, например, вводить в БД данные заведомо неверного типа, дублирующиеся значения первичных ключей и т.п. Набор определенных правил, устанавливающих допустимость значений данных и их связей, называют *правилами* или *ограничениями целостности (constraints).*

Ограничения целостности задаются и хранятся в словаре данных БД как один из элементов определения таблицы, к которой они относятся. Тем самым любое приложение, обращающееся к этой таблице, необходимым образом должно придерживаться заданных правил. Изменения правил целостности может быть произведено на уровне базы данных в целом, а не для отдельного приложения. Это еще один из примеров воплощения принципа независимости данных и прикладного ПО.

Большинство БД подчиняется очень многим правилам поддержки целостности. Есть специфические (корпоративные) правила, которые характерны только для конкретной предметной области и применяется только к одной БД). Есть два общих особых правила, они применяются к любой БД и относятся к потенциальным (и первичным) ключам и ко внешним ключам.

В реляционной модели данных определены два базовых универсальных требования обеспечения целостности:

- целостность сущностей,
- целостность ссылок.

Целостность сущностей

Объект реального мира (сущность) представляется в реляционной базе данных как кортеж некоторого отношения. Требование целостности сущностей заключается в следующем:

каждый объект реального мира должен быть четко идентифицирован, т.е. любое отношение должно обладать первичным ключом.

Вполне очевидно, если данное требование не соблюдается, то в базе данных может храниться противоречивая информация об одном и том же объекте. Поддержание целостности сущностей обеспечивается средствами СУБД. Это осуществляется с помощью двух ограничений:

- при добавлении кортежей проверяется уникальность их первичных ключей и отсутствие в них NULL-значений,
- не позволяется изменение значений атрибутов, входящих в первичный ключ.

При попытке внести любое изменение в базу данных, нарушающее целостность сущностей, операция прерывается, а база данных остается в исходном, согласованном, состоянии.

Целостность ссылок (ссылочная целостность)

Сложные объекты реального мира представляются в реляционной базе данных в виде кортежей нескольких отношений, связанных между собой с помощью внешних ключей.

Требование целостности по ссылкам состоит в следующем:

все значения внешних ключей должны быть *согласованы.*

Выполнение этого требования любая СУБД контролирует автоматически, при этом для каждой из операций манипулирования данными выполняются свои специфические действия. Безусловно, поддержка целостности отнимает много ресурсов, при этом существенно замедляется выполнение операций манипулирования данными, однако гарантированное обеспечение целостности ссылок стоит всех этих затрат.

Правила поддержки ссылочной целостности зависят от выполняемой операции манипулирования данными.

При выполнении *операции вставки* ссылочная целостность контролируется только в случае наличия внешних ключей, ссылающихся на другие отношения. В этом случае проверяется существование соответствующих значений первичных ключей, в случае их отсутствия операция вставки отменяется.

Наиболее остро стоит проблема обеспечения ссылочной целостности при выполнении *операции удаления* кортежей родительского отношения, на которые есть ссылки в дочерних отношениях (возможно, сразу в нескольких). В этом случае простое удаление кортежа приведет к наличию несогласованных значений внешних ключей во всех дочерних отношениях. Обеспечить ссылочную целостность при удалении можно несколькими способами:

1. Запретить удаление кортежей в родительском отношении при наличии хотя бы одного ссылающегося кортежа (restrict - ограничить удаления);
2. При удалении кортежа родительского отношения каскадом удалять все ссылающиеся на него кортежи дочернего отношения (cascade – каскадное удаление);
3. При удалении кортежа родительского отношения установить во всех ссылающихся кортежах NULL-значения во внешних ключах (set null). Этот способ можно применять только в случае, если NULL-значения в соответствующем внешнем ключе разрешены;
4. При удалении кортежа родительского отношения установить во всех ссылающихся кортежах значения по умолчанию во внешних ключах

(set default). Значения по умолчанию задаются при создании базы данных.

Из предложенных способов наиболее безопасным является первый – запрет удаления, он применяется чаще всего. Способ каскадного удаления следует применять очень осторожно.

При выполнении операций обновления во внешних ключах ссылочная целостность обеспечивается так же, как и при добавлении нового кортежа. Обновление первичных ключей при наличии ссылающихся внешних ключей категорически не рекомендуется. Вообще выполнять обновление первичных ключей нет никакой необходимости, даже если СУБД разрешает такую возможность.

2.2. Манипуляционная часть реляционной модели

В реляционной модели определяются два базовых механизма манипулирования данными:

•основанная на теории множеств реляционная алгебра;

•основанное на математической логике реляционное исчисление.

Так же, как и выражения реляционной алгебры, формулы реляционного исчисления определяются над отношениями реляционных баз данных, и результатом вычисления всегда является новое отношение. Новое отношение, полученное из одного или нескольких имеющихся отношений, называется *производным отношением*. Исходные отношения, входящие в состав базы данных, называются *базовыми*.

Реляционная алгебра и реляционное исчисление различаются степенью их процедурности:

•запрос, представленный на языке реляционной алгебры, может быть вычислен на основе вычисления элементарных алгебраических операций с учетом их старшинства и возможных скобок,

•формула реляционного исчисления только устанавливает условия, которым должны удовлетворять кортежи результирующего отношения. Поэтому языки реляционного исчисления являются более непроцедурными или декларативными.

Язык SQL основывается на реляционной алгебре, однако содержит некоторые элементы реляционного исчисления.

2.2.1. Реляционная алгебра

Поскольку отношение в реляционной алгебре определяется как множество кортежей, то на отношения распространяются основные операции над множествами – объединение, пересечение, разность и декарто-

во произведение. Операции объединения, пересечения и разности определены для отношений, схемы которых могут отличаться только именами атрибутов (т.е. отношения имеют одинаковую степень и атрибуты определены на одних и тех же доменах).

1. Объединение R3= R1∪R2

В результат R3 помещаются все кортежи, которые есть в R1 или в R2, причем кортежи, которые одновременно присутствуют в R1 и R2, помещаются в результат один раз.

R1

1	A
2	B
3	C

R2

3	C
4	D
5	E

Результат R3

1	A
2	B
3	C
4	D
5	E

2. Пересечение R3 R1∩R2

В R3 помещаются кортежи, которые есть в R1 и в R2.

Для R1 и R2 из предыдущего примера результатом будет единственный кортеж (3,C).

3. Разность R3=R1-R2

В R3 попадут кортежи, которые есть в R1, но нет в R2

R1

1	A
2	B
3	C

R2

3	C
4	D
5	E

Результат R3

1	A
2	B

Заметим, что операция пересечения может быть вычислена через операции разности R3=R1∩ R2=R1-(R1-R2)

4. Декартово произведение $R3=R1 \times R2$

Результат получается путем склейки каждого кортежа отношения R1 с каждым кортежем R2.

c=a*b

a- количество кортежей в R1

b- количество кортежей в R2

c- количество кортежей в R3

При этом степень R3 равна сумме степеней R1 и R2

R1

1	A
2	B
3	C

R2

3	C
4	D
5	E

Результат R3

1	A	3	C
2	B	3	C
3	C	3	C
1	A	4	D
2	B	4	D
3	C	4	D
1	A	5	E
2	B	5	E
3	C	5	E

Кроме перечисленных операций над множествами, Кодд ввел ряд дополнительных операций над отношениями. Операции проекции и выборки определены над одним отношением R1 (унарные операции).

5. Проекция – отбор атрибутов отношения

$R2=\pi_I (R1)$

I – подмножество атрибутов отношения R1

Степень результата R2 равна $|I|$ - количество элементов в подмножестве I и может принимать целые значения от 1 до степени R1.

Пример: проекция R1 из предыдущих примеров по 2-му атрибуту

Результат R2 содержит всего один атрибут.

A
B
C

6. Операция выборки (селекции) – отбор кортежей

$R2=\sigma_\theta$ (R1)

θ - любое логическое выражение (условие отбора кортежей), в состав которого входят имена атрибутов, операции и константы.

В R2 включают все кортежи из R1, для которых θ - истинно. При этом отношение R2 может не содержать ни одного кортежа или совпадать с отношением R1. Например, при условии отбора

первый атрибут > 0

в R2 попадут все кортежи R1, а при условии

перый атрибут > 2

только один кортеж (3,C)

На практике операции выборки и проекции часто сочетаются в одном запросе.

7. *Операция соединения R3= R1ΘR2*

Данная операция определена над двумя отношениями, у которых есть общее подмножество атрибутов (на практике это чаще всего один общий атрибут, по которому и выполняется операция соединения). В отличие от операции декартова произведения при соединении склеиваются только те кортежи R1 и R2, которые имеют одно и то же значение общего атрибута (а не каждый кортеж с каждым). При этом общий атрибут попадает в результат один раз.

Пример. Пусть выполняется операция соединения по первому из атрибутов (содержащему числовые значения)

R1

1	A
2	B
3	C

R2

2	C
2	D
3	E
4	F

Результат R3

2	B	C
2	B	D
3	C	E

Операция соединения эквивалентна операции выборки из декартова произведения отношений R1 и R2.

$R3= R1\Theta R2=\sigma_\theta (R1\times R2)$

Однако, учитывая большую важность этой операции для реляционной базы данных, где связь между отношениями устанавливается при

помощи общих атрибутов, она включена в состав базовых операций реляционной алгебры.

Следует отметить, что в результат операции соединения не входят кортежи отношений R1 и R2, для которых не находится одинаковых значений в общем атрибуте. Так, в предыдущем примере в результат не вошел кортеж (1,A) из отношения R1 и (4,F) из отношения R2. Ввиду этой особенности данную операцию называют операцией *внутреннего соединения.*

В языке SQL поддерживается три операции *внешнего соединения* – левое, правое и полное.

В левом внешнем соединении результат внутреннего соединения дополняется оставшимися кортежами отношения, стоящего слева (в примере это кортеж (1,A) из отношения R1). Поскольку в результате должно быть 3 атрибута, незаполненный атрибут принимает значение NULL, т.е. в результат R3 добавляется кортеж (1,A,NULL).

В правом внешнем соединении результат внутреннего соединения дополняется оставшимися кортежами отношения, стоящего справа (в примере кортеж (4,F) из отношения R2 дополняется значением NULL и получается кортеж (4,NULL,F) отношения R3).

Наконец, в полном внешнем соединении в результат добавляются все несвязанные кортежи, дополненные неопределенными значениями (в примере это два упомянутых выше кортежа).

8. Операция деления R3 R1÷R2

Для выполнения операции деления отношения R1 и R2 должны иметь общее подмножество атрибутов (обычно один атрибут), причем в отношении R2 это подмножество является множеством его атрибутов (обычно R2 является унарным отношением). Смысл операции поясним на примере.

R1

1	A
1	B
2	C
3	A
3	B
4	A

R2

A
B

В результате получаем отношение

R3

1
3

Операция деления явно не поддерживается в языке SQL, хотя имеется несколько способов выразить ее через другие операции.

2.2.2. Реляционное исчисление

Разницу между реляционной алгеброй и реляционным исчислением поясним на примере.

Пример: Пусть даны два отношения:
СОТРУДНИКИ (СОТР_НОМЕР, СОТР_ИМЯ, СОТР_ЗАРПЛ, ОТД_НОМЕР)
ОТДЕЛЫ(ОТД_НОМЕР, ОТД_КОЛ, ОТД_НАЧ)

Мы хотим узнать имена и номера сотрудников, являющихся начальниками отделов с количеством работников более 10.

Выполнение этого запроса средствами реляционной алгебры распадается на четко определенную последовательность шагов:
1. Выполнить соединение отношений СОТРУДНИКИ и ОТДЕЛЫ по условию СОТР_НОМ = ОТДЕЛ_НАЧ.
С1 = СОТРУДНИКИ [СОТР_НОМ = ОТД_НАЧ] ОТДЕЛЫ

2. Из полученного отношения произвести выборку по условию ОТД_КОЛ>10
С2 = С1 [ОТД_КОЛ > 10].

3. Спроецировать результаты предыдущей операции на атрибуты СОТР_ИМЯ, СОТР_НОМЕР
С3 = С2 [СОТР_ИМЯ, СОТР_НОМЕР]

Заметим, что порядок выполнения шагов может повлиять на эффективность выполнения запроса. Так, время выполнения приведенного выше запроса можно сократить, если поменять местами этапы (1) и (2). В этом случае сначала из отношения СОТРУДНИКИ будет сделана выборка всех кортежей со значением атрибута ОТДЕЛ_КОЛ > 10, а затем выполнено соединение результирующего отношения с отношением ОТДЕЛЫ. Машинное время экономится за счет того, что в операции соединения участвуют меньшие отношения.

На языке реляционного исчисления данный запрос может быть записан как:
Выдать СОТР_ИМЯ и СОТР_НОМ для СОТРУДНИКИ таких, что существует ОТДЕЛ с таким же, что и СОТР_НОМ значением ОТД_НАЧ и значением ОТД_КОЛ большим 50.

Здесь мы указываем лишь характеристики результирующего отношения, но не говорим о способе его формирования. СУБД сама должна решить, какие операции и в каком порядке надо выполнить над отношениями СОТРУДНИКИ и ОТДЕЛЫ. Задача оптимизации выполнения запроса в этом случае также ложится на СУБД.

Рассмотренная система операций реляционной алгебры и ряд предложений реляционного исчисления были приняты в качестве функцио-

нальной спецификации при разработке одной из самых мощных операций языка SQL – операции выборки SELECT, которая используется в подавляющем большинстве современных информационных систем для извлечения и обработки данных, обеспечивая потребности пользователей при решении различных типовых задач автоматизации.

В следующей главе будут рассмотрены теоретические и практические аспекты проектирования реляционных баз данных.

3. Проектирование базы данных

Цель изучения данной главы – освоить общепринятые способы проектирования базы данных.

После изучения главы вы будете:

- знать основные нотации представления диаграмм «сущность - связь»;
- уметь построить диаграмму «сущность - связь» по известной системе бизнес-правил предметной области, используя Case-систему (или без ее использования);
- знать требования всех нормальных форм, уметь выявлять нарушения нормальных форм и выполнять декомпозицию отношений;
- знать преимущества и недостатки нормализации, причины для внесения сознательной денормализации в структуру базы данных;
- иметь представление о хранилищах данных и принципах их организации
- иметь представления об OLTP и OLAP-системах, принципах их организации.

3.1. Семантический анализ предметной области

Предметная область - часть реального мира, подлежащая изучению с целью организации управления и, в конечном счете, автоматизации. Предметная область представляется множеством фрагментов, например, предприятие - цехами, дирекцией, бухгалтерией и т.д. Каждый фрагмент предметной области характеризуется множеством объектов и процессов, использующих объекты, а также множеством пользователей, характеризуемых различными взглядами на предметную область.

3.1.1. Трехуровневая модель ANSI/SPARC

В процессе проектирования базы данных рекомендуется использовать трехуровневую схему описания данных (так называемая трехуровневая модель ANSI/SPARC). Проект трехуровневой модели был выдвинут в 1975 году подкомитетом SPARC (Standards Planning and Requirements Committee) ANSI и имел целью выделить 3 уровня описания данных предметной области, различающихся степенью абстракции (рис.3.1).

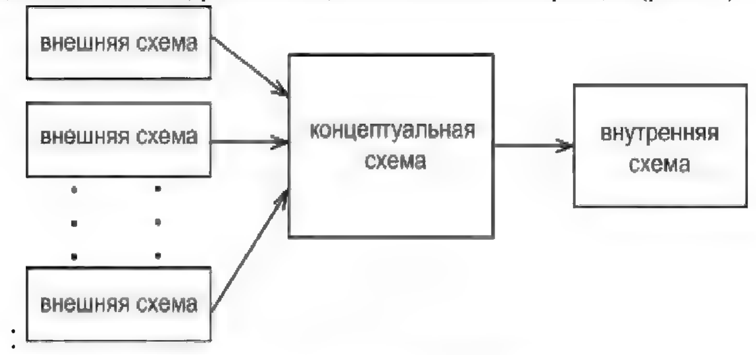


Рис. 3.1. Модель ANSI/SPARC

Внешнее представление (внешняя схема) данных является совокупностью требований к данным со стороны некоторой конкретной функции, выполняемой пользователем. Поскольку пользователей много и их представления о данных предметной области существенно различаются, в процессе анализа предметной области может быть сформировано несколько внешних схем (допустим, данные, которые требуются для сотрудников бухгалтерии, отдела кадров, планово-финансового отдела и т.д.).

Концептуальная схема является полной совокупностью всех требований к данным, полученной из пользовательских представлений о реальном мире. Концептуальная схема описывает все элементы данных и связи между ними, с указанием необходимых ограничений поддержки целостности данных. Для каждой базы данных имеется только одна концептуальная схема. В процессе разработки – это схема проектировщика базы данных, в процессе сопровождения системы – это представление администратора базы данных (АБД).

Важно отметить, что внешние схемы и концептуальная схема относятся к *логическому уровню представления базы данных*, т.е. не учитывают особенностей СУБД, которая используется для создания и поддержки БД – имена типов данных, принятых в СУБД, физические имена файлов, способ хранения данных и.т.д. В связи с этим трудно провести четкую грань между двумя понятиями, применяемыми в теории проектирования

баз данных – *концептуальная схема* и *логическая схема*. Будем считать, что концептуальная схема базы данных – это схема логического уровня, которая содержит полное описание данных предметной области и связей между ними. Все внешние схемы могут быть выведены из концептуальной, но процесс проектирования начинается с внешних схем, поскольку каждый из пользователей владеет только частью информации о предметной области.

Внутренняя схема – это уровень разработчиков СУБД и, частично, АБД и системного администратора. Внутренний уровень описывает физическую реализацию базы данных и предназначен для достижения оптимальной производительности, обеспечения экономного использования дискового пространства, организации мероприятий по защите данных. Он содержит детальное описание структур данных и физической организации файлов с данными, описание вспомогательных структур (индексов), используемых для ускорения поиска, сведения о распределении дискового пространства для хранения данных и индексов, сведения о сжатии данных и выбранных методах их шифрования и т.д. В настоящее время производители СУБД предоставляют АБД довольно много информации о физической организации базы данных, поскольку уровень развития СУБД еще не настолько высок, чтобы все настройки, необходимые для оптимального функционирования базы данных, можно было выполнить автоматически.

Как показывает изучение трехуровневой архитектуры БД, концептуальная схема является самым важным уровнем представления базы данных. Она поддерживает все внешние представления, а сама поддерживается средствами внутренней схемы. Внутренняя схема является всего лишь физическим воплощением концептуальной схемы. Именно концептуальная схема призвана быть полным и точным представлением требований к данным в рамках некоторой предметной области.

Процесс разработки концептуальной схемы БД требует глубокого анализа семантической информации о предметной области. Этот начальный этап проектирования получил название *семантического анализа* предметной области. В результате анализа должны быть определены все элементы данных предметной области в контексте их взаимосвязи с другими данными.

3.1.2. Диаграммы «сущность - связь»

Удобным средством представления концептуальной схемы БД являются *диаграммы «сущность - связь» (entity – relationship diagram, сокращенно ERD)*. Диаграмма «сущность-связь» была предложена в 1976 г. Питером Пин-Шэн Ченом, русский перевод его статьи «Модель "сущ-

ность-связь" - шаг к единому представлению данных» опубликован в журнале «СУБД» N 3 за 1995 г.

Важным для нас является тот факт, что из диаграммы «сущность-связь» могут быть порождены все существующие модели данных (иерархическая, сетевая, реляционная, объектная), поэтому она является наиболее общей. Следует отметить, что современные стандартные способы (нотации) для представления диаграмм «сущность - связь» ближе всего к реляционной модели, а программное обеспечение для моделирования данных обеспечивает автоматическое формирование физической схемы и SQL-сценария создания БД для распространенных реляционных СУБД.

Для пояснения основных принципов построения диаграмм «сущность - связь» будем пользоваться обозначениями, предложенными автором идеи П.Ченом, для обозначения характеристик связей между сущностями воспользуемся обозначениями, предложенными Мартиным.

Далее в примерах диаграмм «сущность - связь» будем использовать международный стандарт IDEF1X.

Элементы диаграммы «сущность - связь»

Любой фрагмент предметной области может быть представлен как множество сущностей, между которыми существует некоторое множество связей. Дадим определения:

Сущность - это объект, который может быть идентифицирован неким способом, отличающим его от других объектов. Сущность имеет множество поименованных свойств (атрибутов) и существует во множестве экземпляров.

Например, выделим несколько сущностей в предметной области *Предприятие*: *Подразделения* (атрибуты *название, руководитель*), *Сотрудники* (атрибуты *личный код, ФИО*), *Дети сотрудников* (атрибуты *имя, дата рождения*), *Проекты с участием сотрудников* (атрибут *Название*).

В нотации П.Чена сущность обозначается прямоугольником, а атрибуты – овалом (рис.3.2).



Рис. 3.2. Обозначения для сущностей и атрибутов

Связь - это ассоциация, установленная между двумя или несколькими сущностями. Особенно часто встречаются бинарные связи, т.е. связи между двумя сущностями.

Роль сущности в связи - функция, которую выполняет сущность в данной связи. Обычно роль обозначается глаголом. Например, в связи *Сотрудники-Дети* сущности сотрудники исполняют роль «является родителем», а дети – «является ребенком». Указание ролей в модели «сущность-связь» не является обязательным и служит для уточнения семантики связи.

Роль сущности в связи в нотации Чена изображается в виде ромба на линии связи. Число экземпляров сущностей, которое может быть ассоциировано через набор связей с экземплярами другой сущности, называют *мощностью связи*.

Могут существовать следующие мощности бинарных связей.

Связь один к одному (обозначается 1 : 1)

Это означает, что в такой связи каждому экземпляру одной сущности всегда соответствует не более одного экземпляра связанной сущности. Так, для сущностей *Подразделения* и *Сотрудники* это связь «руководит», поскольку в каждом подразделении может быть только один начальник, а сотрудник может руководить только одним подразделением. Данный факт представлен на следующем рисунке, где прямоугольники обозначают сущности, а ромб - связь. Так как степень связи для каждой сущности равна 1, то они соединяются одной линией (Рис.3.3).



Рис. 3.3. Связь один к одному между сущностями

Поясним смысл специальных обозначений (вертикальная линия и овал) на линии связи. Они характеризуют *класс принадлежности* входящих в связь сущностей. Так как в каждом подразделении обязательно должен быть руководитель, то каждому экземпляру сущности *Подразделения* непременно должен соответствовать экземпляр сущности *Сотрудники*. Однако не каждый сотрудник является руководителем отдела, следовательно, в данной связи не каждый экземпляр сущности *Сотрудники* имеет связанный экземпляр сущности *Подразделения*.

Таким образом, сущность *Сотрудники* имеет *обязательный класс принадлежности* (этот факт обозначается вертикальной линией), а сущность *Подразделения* имеет *необязательный класс принадлежности* (обозначается овалом).

Связь один ко многим (1 : М)

В данном случае экземпляру сущности может соответствовать любое число экземпляров другой сущности. Такова связь *Подразделения – Сотрудники,* при условии, что каждый сотрудник может работать только в одном подразделении (на рис. 3.4 показаны две различных связи между *Сотрудниками* и *Подразделениями*).

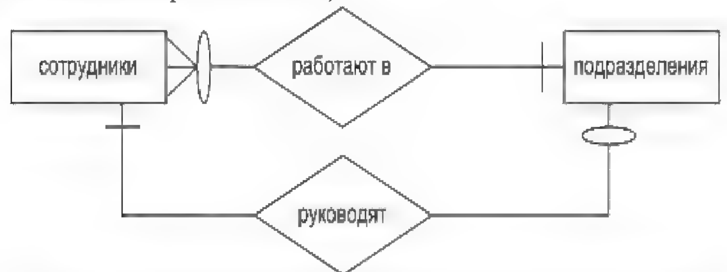


Рис. 3.4. Связи между сущностями Сотрудники и Подразделения

Данный рисунок дополнительно иллюстрирует тот факт, что между двумя сущностями может быть определено несколько связей. Здесь также необходимо учитывать класс принадлежности сущностей. Каждый сотрудник должен работать в каком-либо отделе, но не каждый отдел (например, вновь сформированный) должен включать хотя бы одного сотрудника. Поэтому сущность *Подразделения* имеет обязательный, а сущность *Сотрудники* необязательный классы принадлежности.

Связь много к одному (М : 1) Эта связь аналогична отображению *1 :М.*

Связь многие ко многим (М :М)

В этом случае каждая из связанных сущностей может быть представлена любым количеством экземпляров. Пусть сотрудники на рассматриваемом нами предприятии в процессе своей трудовой деятельности участвуют в различных проектах. Поскольку каждый сотрудник может участвовать в нескольких (в том числе и ни в одном) проектах, а каждый проект может выполняться сразу целой группой сотрудников, то связь между сущностями *Сотрудники* и *Проекты* имеет степень *М : М* (рис.3.5).



Рис. 3.5. Связь М:М между Сотрудниками и Проектами

В заключение приведем небольшой фрагмент диаграммы «сущность - связь» для проанализированных сущностей предметной области *Предприятие* в нотации П.Чена (с дополнениями Мартина). Он соответствует рассмотренным выше бизнес-правилам предприятия.

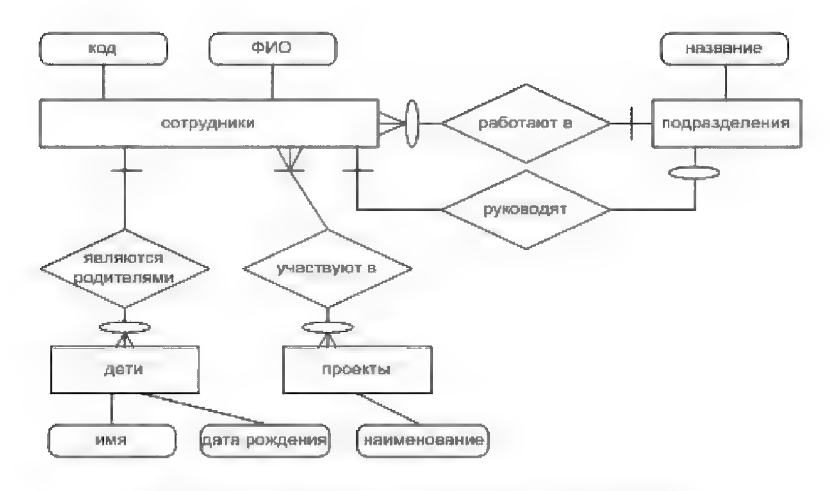


Рис. 3.6. Фрагмент диаграммы «сущность - связь»

Безусловно, реальная подсистема учета персонала предприятия требует анализа еще очень многих сущностей и связей.

3.1.3. CASE-технологии и CASE-системы

Современные информационные системы имеют очень высокую сложность и хранят огромное количество данных. Например, известная система дистанционного обучения Moodle содержит базу данных более чем из 200 таблиц (причем в каждой новой версии появляется по несколько новых таблиц), а ведь эта система считается системой средней сложности. Интегрированные системы предприятий могут содержать и до 1000 таблиц.



Рис. 3.7. Этапы жизненного цикла информационной системы

Для автоматизации столь трудоемкого процесса, как анализ предметной области и разработка концептуальной схемы базы данных, требуется особая технология. Такая технология получила название CASE (*Computer Aided Software Engeneering* - создание программного обеспечения с помощью компьютера). Основные черты CASE - технологии:

- разработка информационной системы представляется в виде последовательных четко определенных этапов (рис. 3.7),

- поддержка всех этапов жизненного цикла ИС, начиная с анализа предметной области до получения и сопровождения готового программного продукта,
- поддержка репозитария, хранящего спецификации проекта ИС на всех этапах ее разработки,
- возможность одновременной работы с репозитарием многих разработчиков,
- автоматизация различных стандартных действий по проектированию и реализации ИС.

Как правило, CASE-системы поддерживают следующие этапы процесса разработки информационной системы.

- Моделирование и анализ деятельности пользователей в рамках предметной области. Здесь осуществляется функциональная декомпозиция, определение иерархий (вложенности) функций, построение диаграмм потоков данных. Перечень информационных объектов, которыми манипулируют функции, передается на следующий этап проектирования.
- Концептуальное моделирование - создание диаграммы "сущность-связь" на основе перечня объектов, полученного на предыдущем этапе.
- Преобразование диаграммы "сущность-связь" в физическую схему базы данных, учитывающую особенности выбранной СУБД. Это преобразование выполняется Case-системой автоматически .
- Автоматическая генерация SQL-сценария создания базы данных. Результатом выполнения данного этапа является набор SQL-операторов, описывающих создание схемы базы данных с учетом особенностей выбранной СУБД.
- Некоторые Case-системы выполняют генерацию прототипов программных модулей прикладного программного обеспечения, заготовки экранных форм и отчетов.

В настоящее время имеется большое количество CASE-систем, поддерживающих разные нотации изображения диаграмм «сущность - связь». Далее рассмотрим одну из наиболее популярных нотаций и основанную на ней методологию IDEF1.

3.1.4. Методология IDEF1

Метод IDEF1, разработанный Т. Рэмей (T.Ramey), основан на подходе П. Чена. В настоящее время на основе совершенствования методологии IDEF1 создана ее новая версия - методология IDEF1X. IDEF1X разработана с учетом таких требований, как простота изучения и возможность автоматизации. IDEF1X-диаграммы используются рядом распространен-

ных CASE-систем, в частности, это ERwin Data Modeller, Design/IDEF, свободно распространяемая система TOAD Data Modeller и ряд других.

Сущность, как в подходе Чена, обозначается прямоугольником. Список атрибутов приводится внутри прямоугольника, обозначающего сущность. Атрибуты, составляющие ключ сущности, группируются в верхней части прямоугольника и отделяются горизонтальной чертой.

Связь изображается линией, проводимой между сущностью-родителем и дочерней сущностью точкой на конце линии у дочерней сущности. Дополнительно может определяться мощность связи (количество экземпляров дочерней сущности, которое может существовать для каждого экземпляра сущности-родителя). В IDEF1X могут быть выражены следующие мощности связей:

- каждый экземпляр сущности-родителя может иметь ноль, один или более связанных с ним экземпляров дочерней сущности;
- каждый экземпляр сущности-родителя должен иметь не менее одного связанного с ним экземпляра дочерней сущности;
- каждый экземпляр сущности-родителя должен иметь не более одного связанного с ним экземпляра дочерней сущности;
- каждый экземпляр сущности-родителя связан с некоторым фиксированным числом экземпляров дочерней сущности.

Если экземпляр дочерней сущности однозначно определяется своей связью с сущностью-родителем, то связь называется идентифицирующей, в противном случае - неидентифицирующей.

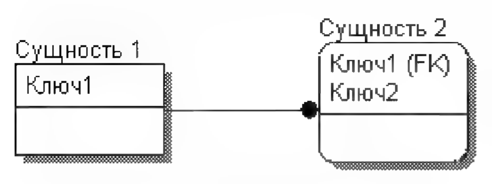


Рис. 3.8. Идентифицирующая связь

Идентифицирующая связь между сущностью-родителем и сущностью-потомком изображается сплошной линией (рис. 3.8). Сущность-потомок в идентифицирующей связи является *зависимой сущностью* (изображается на диаграмме прямоугольником с закругленными концами).

В приведенном примере Сущность2 имеет составной первичный ключ (Ключ1, Ключ2), т.е. сущность2 не имеет собственного идентификатора, а идентифицируется через первичный ключ родителя.

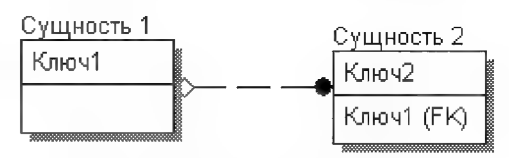


Рис. 3.9. Неидентифицирующая связь

Пунктирная линия изображает неидентифицирующую связь (рис. 3.9). Сущность-потомок в неидентифицирующей связи

будет независимой от ключа родителя, если она не является также сущностью-потомком в какой-либо идентифицирующей связи. Неидентифицирующая связь является более слабой, чем идентифицирующая, а сущность-потомок – более независимой от родителя.

Некоторые CASE-системы, например ER-Win, позволяют изображать на диаграмме связь «многие-ко-многим» в виде сплошной линии с точками на обоих концах (рис.3.10), при этом выполняют автоматическое формирование ассоциированной сущности, которая в физической схеме превращается в таблицу-связку.

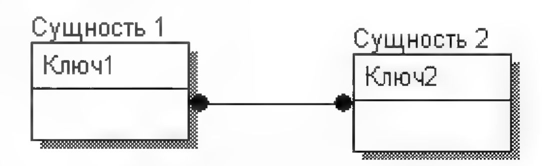


Рис. 3.10. Связь Многие ко многим

В заключение приведем фрагмент диаграммы «сущность-связь», изображенный на рис.3.6, в нотации IDEF1X (рис.3.11).

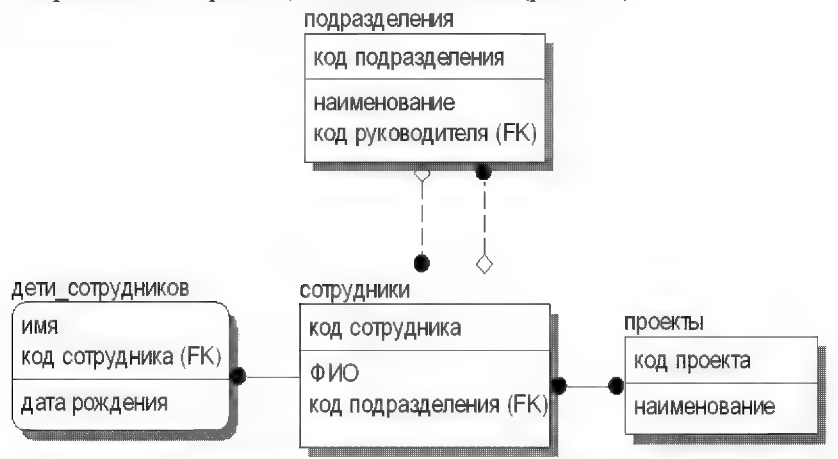


Рис. 3.11. Фрагмент диаграммы «сущность-связь» (IDEF1X)

Здесь следует обратить внимание на связи между *подразделениями* и *сотрудниками*. Связь слева имеет мощность 1:М (в каждом подразделении много сотрудников), связь справа имеет мощность 1:1 (каждый сотрудник может руководить не более чем одним подразделением). Но обе связи являются необязательными, т.е. сотрудник может не руководить никаким подразделением, а подразделение может какое-то время существовать без сотрудников.

3.2. Нормализация базы данных

Диаграммы «сущность – связь» являются самым общим способом описания данных предметной области, который базируется только на семантическом анализе информации и не учитывает особенностей реляционной модели данных.

Современные методологии, такие как IDEF1X, уже используют терминологию реляционных баз данных (например, понятия внешнего ключа, ассоциированной сущности и т.д.) и позволяют проектировать модель «сущность – связь», которая затем автоматически преобразуется в схему реляционной базы данных. Однако процесс выделения сущностей и анализа связей по-прежнему остается не формализованным, а результаты существенно зависят от опыта и аналитических способностей проектировщика.

В процессе разработки основ реляционной теории был предложен формальный математический аппарат, позволяющий проектировать реляционные базы данных с минимальной избыточностью, который получил название *нормализации* базы данных. В настоящее время нормализация не рассматривается как основной аппарат проектирования БД, но является прекрасным средством анализа имеющейся схемы БД (например, полученной с помощью методологии IDEF1X) с целью обнаружения и устранения избыточности данных.

Нормализация основана на анализе *функциональных зависимостей (ФЗ)* между атрибутами отношений. Начнем с формального определения ФЗ и их математических свойств.

3.2.1. Определение функциональной зависимости

Если даны два атрибута X и Y некоторого отношения, то говорят, что Y функционально зависит от X, если в любой момент времени каждому значению X соответствует ровно одно значение Y.

Функциональная зависимость обозначается X → Y. Отметим, что X и Y могут представлять собой не только единичные атрибуты, но и группы (подмножества), составленные из нескольких атрибутов одного отношения.

Подмножество атрибутов X называют детерминантом функциональной зависимости.

Можно сказать, что функциональные зависимости представляют собой связи типа "один ко многим", существующие внутри отношения.

Например, в отношении *Сотрудники (код, ФИО, пол, дата_рождения)* можно выделить довольно много ФЗ. Вот некоторые из них:

- *код → ФИО*
- *код → пол*

- *код → дата_рождения*
- *код → (ФИО, дата_рождения),*

поскольку каждому значению кода соответствует ровно одно значение атрибутов ФИО, пол и дата_рождения и любых их сочетаний. Однако нельзя с уверенностью сказать, что каждому значению ФИО соответствует ровно одно значение кода (могут оказаться полные однофамильцы), поэтому нельзя говорить о наличии ФЗ *ФИО → код*. Тем более, не может быть, например, ФЗ *дата_рождения → ФИО*.

Атрибуты ФИО и дата_рождения являются *взаимно независимыми*, поскольку в общем случае ни один из них не зависит от другого. Заметим, что если текущее состояние базы данных таково, что все ФИО различны, все равно нельзя говорить о наличии ФЗ *ФИО → дата рождения,* поскольку уникальность и/или неизменность столбца ФИО явно не декларирована и значения-дубликаты могут появиться в любой момент.

Стоит обратить внимание на возможность существования ФЗ *ФИО → пол.* Создание программного кода, которое может автоматически безошибочно вычислять пол по заданному значению ФИО, весьма проблематично при существующем разнообразии способов формирования ФИО и национальных особенностях. Поэтому в существующих БД принято пол хранить явно и считать, что такой ФЗ в отношении нет.

3.2.2. Математические свойства ФЗ, теоремы

Рассмотрим некоторые математические свойства ФЗ, вытекающие из ее определения. Данные свойства получили название *правил Армстронга* по имени исследователя, который их сформулировал.

1. Рефлективность

Если Y является подмножеством X, то X определяет Y

$Y \subset X => X \to Y$

Такая функциональная зависимость называется тривиальной.

2. Дополнение

$A \to B$ => $AC \to BC$, где C – любое подмножество атрибутов отношения.

3. Транзитивность

$A \to B$ и $B \to C$ => $A \to C$

В этом случае говорят, что C *транзитивно зависит* от A. К этому понятию мы еще вернемся при рассмотрении третьей нормальной формы.

Из этих трех основных свойств можно вывести еще несколько.

4. Самоопределение

$X \to X$

Такая зависимость не несет какой-либо информации, однако она удовлетворяет определению ФЗ

5. Декомпозиция
$A \to BC \Rightarrow A \to B, A \to C$

6. Композиция
$A \to B$ и $C \to D \Rightarrow AC \to BD$

7. Теорема о всеобщей зависимости или теорема всеобщего объединения

Если $A \to B$ и $C \to D \Rightarrow A \cup (C-B) \to BD$

Здесь $\cup$ - операция объединения множеств.

Правила Армстронга позволяют выводить новые ФЗ на основе других ФЗ. Применяя их, можно вывести *множество всех ФЗ данного отношения.* Такое множество называется *замыканием.*

И наоборот, для каждого отношения можно найти такое множество *всех* ФЗ, в котором ни одна ФЗ не может быть выведена из другой. Такое множество ФЗ заданного отношения называется *неприводимым.* К.Дж.Дейт показал, что неприводимое множество ФЗ должно обладать следующими свойствами:

- в правой части каждой ФЗ должен быть только одни атрибут;
- из левой части каждой ФЗ нельзя удалить ни одного атрибута без потери этой ФЗ

Такое множество для каждого отношения может быть только одно.

Возвращаясь к примеру с ФЗ отношения *Сотрудники (код, ФИО, пол, дата_рождения),* отметим, что приведенное в примере множество ФЗ

- *код → ФИО*
- *код → пол*
- *код → дата_рождения*
- *код → (ФИО, пол)*

не является неприводимым, т.к. последняя ФЗ может быть легко выведена из первых двух. Первые три ФЗ составляют неприводимое множество ФЗ для отношения *Сотрудники*.

Приведенное множество не является и замыканием, поскольку из приведенных выше ФЗ можно вывести еще много других ФЗ, например, *(код, ФИО) → дата_рождения, (код, дата_рождения) → ФИО* и т.д.

3.2.3. Процедура нормализации. Декомпозиция отношений

Нежелательные ФЗ

Некоторые функциональные зависимости, входящие в неприводимое множество ФЗ отношения, являются нежелательными, так как приводят к дублированию информации, что, в свою очередь, создает проблемы обновления данных и представляет угрозу целостности БД.

Например, добавим в отношение *Сотрудники* дополнительный атрибут *возрастная категория,* который может принимать значения «несовершеннолетний», «пенсионер» и NULL для остальных сотрудников. Добавление этого атрибута внесет новую ФЗ в неприводимое множество ФЗ:

(дата_рождения, пол) → возрастная_категория

(граница пенсионного возраста зависит от пола, поэтому в общем случае возрастную категорию определяют два атрибута)

Данную ФЗ отнесем к нежелательным по двум причинам. Во-первых, новый столбец *возрастная_категория* будет содержать очень много дублирующихся значений. Во-вторых, для поддержки актуального состояния этого атрибута придется каждый день выполнять пересчет его значений. Если хотя бы один день не будет выполнено такое обновление, база данных окажется в противоречивом состоянии. При любых законодательных изменениях возрастных границ снова придется обновлять атрибут *возрастная_категория* для всех сотрудников.

Правильным решением будет создание отдельного отношения, содержащего всю информацию, необходимую для определения возрастной категории по известной дате рождения и полу сотрудника. Можно предложить, например, такую схему отношения:

Возрастные_категории (код_категории, название, пол, возрастная граница)

В отношении будет всего три кортежа (по одному на каждую возрастную категорию), поэтому никакого дублирования информации не будет. Любые изменения возрастных границ достаточно обновить в одном месте.

Приведенный пример пока не дает ответа на вопрос, каким образом удобнее выявлять нежелательные ФЗ, однако указывает путь их устранения – вынесение нежелательных ФЗ в отдельные отношения.

Этот процесс называется *декомпозицией* (разбиением) имеющихся отношений. В процессе декомпозиции количество отношений в базе данных увеличивается, но общее количество хранимых данных, как правило, сокращается за счет устранения дублирования данных.

Декомпозиция без потерь. Теорема Хеза

Декомпозиция отношения есть не что иное, как взятие одной или нескольких проекций исходного отношения так, чтобы эти проекции в совокупности содержали (возможно, с повторениями) *все* атрибуты исходного отношения. Иначе говоря, при декомпозиции *не должны теряться атрибуты* отношений. Но при декомпозиции также не должны потеряться и сами данные. Данные можно считать не потерянными в том случае, если по декомпозированным отношениям можно полностью восстановить исходное отношение *в прежнем виде,* используя операцию соединения отношений.

Проекции R_1 и R_2 отношения R называются *декомпозицией без потерь*, если отношение *R точно восстанавливается* из них при помощи естественного соединения *для любого состояния* отношения *R*.

Теорема Хеза

Пусть $R(A, B, C)$ - отношение, A, B, C - атрибуты или множества атрибутов этого отношения. Если имеется функциональная зависимость $A \rightarrow B$, то проекции $R_1(A, B)$ и $R_2(A, C)$ образуют декомпозицию без потерь.

Иными словами, *любую ФЗ отношения можно вынести в отдельное отношение, оставив ее детерминант в исходном отношении.* При этом никакая информация не будет утеряна.

Обратимый пошаговый процесс декомпозиции отношений с устранением нежелательных функциональных зависимостей называется *нормализацией*.

3.2.4. Нормальные формы

Процесс нормализации выполняется в несколько этапов, на каждом из этапов база данных приводится к очередной *нормальной форме*, причем каждая следующая нормальная форма обладает свойствами лучшими, чем предыдущая.

Каждой нормальной форме соответствует некоторый определенный набор ограничений, и отношение находится в некоторой нормальной форме, если удовлетворяет свойственному ей набору ограничений.

В теории реляционных баз данных обычно выделяется следующая последовательность нормальных форм:

- первая нормальная форма (1NF);
- вторая нормальная форма (2NF);
- третья нормальная форма (3NF);
- нормальная форма Бойса-Кодда (BCNF);
- четвертая нормальная форма (4NF);

- пятая нормальная форма, или нормальная форма проекции-соединения (5NF или PJ/NF).

Основные свойства нормальных форм:

- каждая следующая нормальная форма в некотором смысле лучше предыдущей;
- при переходе к следующей нормальной форме свойства предыдущих нормальных свойств сохраняются.

На практике обычно доводят базу данных до третьей нормальной формы. Подробный анализ положительных и отрицательных сторон нормализации содержится в следующей лекции.

Рассмотрим подробнее каждый из этапов нормализации.

Требования первой нормальной формы - 1NF

Отношение находится в 1NF, если оно удовлетворяет двум условиям:

1. Значения всех его атрибутов атомарны;
2. Отсутствуют повторяющиеся группы атрибутов.

Первое из условий выполняется для любого отношения автоматически, поскольку оно следует из свойств отношений. Здесь требуется только уточнить понятие атомарности для текстовых атрибутов. Например, такие атрибуты, как ФИО или адрес_сотрудника, можно считать атомарными только в тех случаях, когда *все* запросы к базе данных требуют извлечения этих атрибутов *целиком*, без вычленения отдельных составляющих.

В подавляющем большинстве случаев ФИО хранится в виде трех отдельных атрибутов-атомов, адрес также разбивается на несколько составляющих, каждая из которых в рамках бизнес-правил предметной области является неделимым атомом. В рамках данной лекции будем считать ФИО атомарным атрибутом.

Второе условие требует отдельного пояснения. *Повторяющейся группой* называют подмножество атрибутов отношения, определенных на одном и том же домене и имеющих одинаковую семантику. Например, рассмотрим отношение

Доходы_сотрудников

(код, ФИО, доход_за_январь, за_февраль,...,за_декабрь)

Здесь явно видна повторяющаяся группа из 12 атрибутов, каждый из которых хранит доходы сотрудников за различные месяцы. Такая таблица удобна для формирования отчетов и справок о доходах, она не содержит нежелательных ФЗ, однако многие операции реляционной алгебры не рассчитаны на такую структуру, поэтому многие запросы будут сложно реализовать. В данном случае устранить повторяющуюся группу несложно, поскольку каждый из месяцев имеет известный порядковый номер. Изменим схему отношения:

Доходы_сотрудников (код, ФИО, номер месяца, доход)

Количество кортежей в новом отношении будет в 12 раз больше, чем в исходном, но строить произвольные запросы к такому отношению гораздо удобнее. Данное отношение удовлетворяет требованиям *первой нормальной формы.*

К сожалению, выполнив преобразование отношения, мы внесли одну серьезную проблему – значения атрибута ФИО будут дублироваться в каждом из 12 кортежей, относящихся к одному сотруднику.

На следующем этапе нормализации эта проблема будет решена.

Требования второй нормальной формы - 2NF

Первичный ключ отношения иногда включает несколько атрибутов (в таком случае его называют составным). Например, отношение *Доходы_сотрудников* после преобразования имеет составной ключ

(*код, номер_месяца*).

Введем понятие полной функциональной зависимости.

Определение: неключевой атрибут функционально полно зависит от составного ключа, если он функционально зависит от всего ключа в целом, но не находится в функциональной зависимости от какого-либо подмножества составного ключа.

Рассмотрим неприводимое множество функциональных зависимостей для отношения *Доходы сотрудников:*

(код, месяц) $\rightarrow$ *доход*

код $\rightarrow$ *ФИО*

Первая ФЗ является полной, вторая ФЗ имеет в качестве детерминанта подмножество составного ключа. Именно эта ФЗ создает дублирование значений атрибута ФИО, поэтому является *нежелательной.* Дублирование информации создает и проблемы обновления: в случае необходимости изменения ФИО (допустим, сотрудник обнаружил ошибку в записи своей фамилии), исправления придется вносить 12 раз.

Для устранения нежелательной ФЗ выполним декомпозицию в соответствии с теоремой Хеза – выделим эту ФЗ в отдельное отношение. В итоге получим нормализованную структуру из двух отношений:

Сотрудники (код, ФИО)

Доходы_сотрудников (код, номер месяца, доход)

Для этого примера процесс нормализации завершен – ни одно из отношений больше не содержит нежелательных ФЗ. Дублирование информации устранено.

Таким образом, можно дать определение второй нормальной формы:

Отношение находится в 2NF, если оно находится в 1NF и каждый неключевой атрибут функционально полно зависит от ключа.

Требования третьей нормальной формы - 3NF

Перед обсуждением третьей нормальной формы напомним понятие транзитивной ФЗ, которое уже рассматривалось выше.

Определение: Пусть X, Y, Z - три атрибута некоторого отношения. При этом X $\rightarrow$ Y и Y $\rightarrow$ Z. Тогда Z транзитивно зависит от X.

Рассмотрим еще одну возможную схему отношения *Сотрудники (код, ФИО, должность, должностной оклад).* Запишем неприводимое множество функциональных зависимостей при условии, что должностной оклад сотрудника зависит *только* от занимаемой должности (предположим, что это правило согласовано с представителями бизнеса):

код $\rightarrow$ *ФИО*

код $\rightarrow$ *должность*

должность $\rightarrow$ *должностной оклад*

Получается, что должностной оклад зависит от кода сотрудника транзитивно, через должность. ФЗ *должность* $\rightarrow$ *должностной оклад* в рассматриваемом отношении является *нежелательной* и приводит к дублированию значений атрибута *должностной оклад.*

При этом возникают следующие проблемы обновления данных:

• если в данный момент ни один из сотрудников не работает в какой-либо должности (например, психолог), нельзя ввести данные о соответствующем должностном окладе,

• при изменении должностных окладов необходим просмотр всего отношения и изменение кортежей для всех сотрудников, которые занимают должности с изменившимся окладом.

Для устранения этих проблем необходимо декомпозировать исходное отношение на два:

Сотрудники (код, ФИО, должность) и

Должности (должность, должностной оклад).

Определение третьей нормальной формы:

Отношение находится в 3NF, если оно находится в 2NF и *каждый неключевой атрибут нетранзитивно зависит от первичного ключа.*

Нормальная форма Бойса-Кодда - BCNF

Эта нормальная форма вводит дополнительное ограничение по сравнению с третьей нормальной формой.

Определение нормальной формы Бойса-Кодда:

Отношение находится в BCNF, если оно находится в 3NF и в ней отсутствуют зависимости атрибутов первичного ключа от неключевых атрибутов.

Ситуация, когда отношение находится в 3NF, но не в BCNF, возникает при условии, что отношение имеет два (или более) возможных ключа, которые являются составными и имеют общие атрибуты.

Заметим, что на практике такая ситуация встречается достаточно редко, для всех прочих отношений 3NF и BCNF эквивалентны.

Требования четвертой нормальной формы - 4NF

Четвертая нормальная форма касается отношений, в которых имеются повторяющиеся наборы данных. На практике нарушения 4NF часто происходят в таблицах-связках, которые связывают более двух сущностей. В этом случае используют декомпозицию, основанную на многозначных зависимостях.

Многозначная зависимость является обобщением функциональной зависимости.

В качестве примера рассмотрим отношение:

Расписание_занятий (*день_недели, номер_пары, код_группы, предмет, преподаватель, аудитория*),

хранящее сведения о расписании занятий студенческих групп за неделю (для упрощения считаем, что нет разбиения на подгруппы и объединения в потоки).

В данном отношении можно выделить несколько альтернативных составных ключей:

(*день_недели, номер_пары, код_группы*),

(*день_недели, номер_пары, преподаватель*),

(*день_недели, номер_пары, аудитория*).

Некоторые атрибуты этого отношения трудно назвать абсолютно независимыми. Однако при условии, что каждый преподаватель может вести несколько предметов, а каждый предмет может вести несколько преподавателей, между этими атрибутами отсутствует ФЗ, соответствующая определению, приведенному в начале лекции. Если предположить, что аудитория явно не определяется ни предметом, ни группой, ни преподавателем, никаких нежелательных ФЗ в отношении выделить нельзя, поэтому оно удовлетворяет требованиям не только 3NF, но и NFBC.

Связь между атрибутами *предмет* и *преподаватель* можно определить как «многие ко многим». Между ними существует *многозначная зависимость,* которая обозначается так:

предмет ->> *преподаватель*

Аналогичная многозначная зависимость существует между атрибутами *предмет* и *группа,* поскольку каждая студенческая группа имеет определенный учебный план и изучает определенное заранее множество предметов.

Наличие многозначных зависимостей привносит в отношение значительную избыточность и приводит к возникновению проблем с обновлением данных. Например, если какой-то преподаватель уволился и его место занял другой, то придется обновить все кортежи *расписания,* в которых присутствует данный преподаватель. Кроме того, имеется опасность внести в расписание предмет, который не входит в учебный план группы или преподавателя, который не ведет данный предмет.

Определение четвертой нормальной формы:

Отношение находится в 4NF, если оно находится в BCNF и в нем отстутсвуют многозначные зависимости.

В приведенном примере не так-то просто избавиться от многозначных зависимостей. Можно предложить такой вариант:

Выносим в отдельные отношения зависимости *предмет* и *преподаватель* и *предмет* и *группа.* Естественное соединение этих отношений даст допустимые сочетания (*предмет, преподаватель, группа).*

Тогда схема отношения *Расписание* будет иметь вид:

Расписание (день_недели, номер_пары, аудитория, код допустимого сочетания предмет_ преподаватель_ группа)

Требования пятой нормальной формы - 5NF

До сих пор мы предполагали, что единственной операцией, необходимой для устранения избыточности в отношении, была декомпозиция его на две проекции. Однако, существуют отношения, для которых нельзя выполнить декомпозицию без потерь на две проекции, но которые можно подвергнуть декомпозиции без потерь на три (или более) проекций. Этот факт получил название зависимости по соединению, а такие отношения называют 3-декомпозируемые отношения (ясно, что любое отношение можно назвать "n-декомпозируемым", где n >= 2).

Зависимость по соединению является обобщением многозначной зависимости. Отношения, в которых имеются зависимости по соединению, не являющиеся одновременно ни многозначными, ни функциональными, также характеризуются аномалиями обновления. Поэтому вводится понятие пятой нормальной формы.

Определение пятой нормальной формы:

Отношение находится в 5NF тогда и только тогда, когда любая зависимость по соединению в нем определяется только его потенциальными ключами.

На практике требования 5NF проверить достаточно сложно, а избавление от зависимостей соединения резко увеличивает число отношений в базе данных. В силу этих причин пятая нормальная форма представляет скорее теоретический интерес, чем практическую потребность.

3.3. Денормализация. Хранилища данных

3.3.1. Недостатки нормализованной базы данных

До сих пор речь шла только о недостатках ненормализованных (или в недостаточной мере нормализованных) баз данных, причем в качестве основных недостатков отмечались:

- неэкономное использование дискового пространства ввиду наличия дублирующихся данных;
- проблемы, связанные с обновлением данных ввиду нарушения фундаментального принципа «Каждый факт – в одном месте»;
- связанные с ними проблемы усиленного контроля целостности данных, которая подвергается серьезным испытаниям при наличии дублирующихся данных.

Можно сделать вывод, что для динамично обновляющихся баз данных нормализация является необходимостью, позволяющей избежать многих проблем в процессе функционирования ИС.

Тем не менее, нормализация имеет один существенный недостаток - замедление работы СУБД при выполнении запросов на извлечение (выборку) данных. В нормализованной базе данных практически каждый запрос требует соединения данных нескольких таблиц (иногда в запросах приходится соединять и довольно много таблиц). Соединение таблиц – операция, требующая определенных затрат ресурсов – памяти, процессорного времени. Чем выше степень нормализованности базы данных, тем больше в ней таблиц, следовательно, тем медленнее выполняются запросы на выборку.

В силу этих обстоятельств на практике обычно стараются найти разумный компромисс и редко доводят нормализацию до 5NF. Практика показала, что большинство динамично обновляющихся БД обычно доводится до 3NF. Во многих случаях полезно избавление от многозначных зависимостей (4NF).

Если количество запросов на выборку существенно превышает количество запросов на обновление, имеет смысл проанализировать возможности сознательной *денормализации* базы данных и поступиться даже требованиями 3NF.

Денормализация это процесс модификации структуры таблиц нормализованной базы данных с целью повышения производительности за счет допущения некоторой управляемой избыточности данных. *Единственным оправданием денормализации является попытка повышения скорости работы приложений, работающих с базой данных.*

Денормализованная база данных — это не то же самое, что ненормализованная. *Денормализация* базы данных представляет собой процесс понижения нормализации на один-два уровня.

Денормализация предполагает объединение некоторых из ранее разделенных таблиц и создание таблиц с дубликатами данных с целью уменьшения числа связываемых таблиц при доступе к данным, что должно уменьшить число требуемых операций ввода-вывода и нагрузку на центральный процессор.

Однако за денормализацию нужно платить. В денормализованной базе данных повышается избыточность данных, что может повысить производительность, но потребует больше усилий для контроля ссылочной целостности. Усложнится процесс создания приложений, поскольку данные будут повторяться и их труднее будет отслеживать.

Существует золотая середина между нормализацией и денормализацией, но чтобы найти ее, требуются немалые усилия и хорошее знание предметной области.

К сожалению, не существует никаких формализованных методов для достижения удовлетворительного компромисса между нормализацией и денормализацией. Некоторые соображения по этому поводу содержатся в следующем разделе.

3.3.2. OLTP и OLAP-системы. Data Mining

Можно выделить некоторые классы информационных систем, для которых больше подходят сильно или слабо нормализованные модели данных.

Сильно нормализованные модели данных хорошо подходят для так называемых *OLTP-систем* (*On-Line Transaction Processing* - *оперативная обработка транзакций*).

Типичными примерами OLTP-систем являются системы складского учета, системы заказов билетов, банковские системы, выполняющие операции по переводу денег, и т.п. Основная функция подобных систем заключается в выполнении большого количества коротких *транзакций*. Механизм транзакций будет подробно рассмотрен в лекции 16, для понимания принципов работы *OLTP-систем* достаточно представлять транзакцию как атомарное действие, изменяющее состояние базы данных.

Транзакции в OLTP-системе являются относительно простыми, например, «снять сумму денег со счета А и добавить эту сумму на счет В». Проблема заключается в том, что, во-первых, транзакций очень много, во-вторых, выполняются они одновременно (к системе может быть подключено несколько тысяч одновременно работающих пользователей), в-третьих, при возникновении ошибки, транзакция должна целиком откатиться и вернуть систему к состоянию, которое было до начала транзакции (не должно быть ситуации, когда деньги сняты со счета А, но не поступили на счет В).

Практически все запросы к базе данных в OLTP-приложениях состоят из команд вставки, обновления, удаления. Запросы на выборку в основном предназначены для предоставления пользователям возможности выбора из различных справочников. Большая часть запросов известна заранее еще на этапе проектирования системы. Таким образом, критическим для OLTP-приложений является скорость и надежность выполнения коротких операций обновления данных.

База данных, с которой работают OLTP-приложения, постоянно обновляется, в связи с этим ее обычно называют *оперативной БД*. Чем выше уровень нормализации оперативной БД, тем быстрее и надежнее работают OLTP-приложения. Отступления от этого правила могут происходить тогда, когда уже на этапе разработки известны некоторые часто возникающие запросы, требующие соединения отношений и от скорости выполнения которых существенно зависит работа приложений. В этом случае можно сознательно внести некоторую избыточность в базу данных для ускорения выполнения подобных запросов.

Другим типом информационных систем являются так называемые *OLAP-системы* (*On-Line Analitical Processing - оперативная аналитическая обработка данных*). OLAP используется для принятия управленческих решений, поэтому системы, использующие технологию OLAP, называют *системами поддержки принятия решений* (*Decision Support System - DSS*).

Концепция OLAP была описана в 1993 году Эдгаром Коддом, автором реляционной модели данных. В 1995 году на основе требований, изложенных Коддом, был сформулирован так называемый *тест FASMI (Fast Analysis of Shared Multidimensional Information* — быстрый анализ разделяемой многомерной информации), включающий следующие требования к приложениям для многомерного анализа:

- предоставление пользователю результатов анализа за приемлемое время (обычно не более 5 с), пусть даже ценой менее детального анализа;
- возможность осуществления любого логического и статистического анализа, характерного для данного приложения, и его сохранения в доступном для конечного пользователя виде;
- многопользовательский доступ к данным с поддержкой соответствующих механизмов блокировок и средств авторизованного доступа;
- многомерное концептуальное представление данных, включая полную поддержку для иерархий и множественных иерархий (это — ключевое требование OLAP);
- возможность обращаться к любой нужной информации независимо от ее объема и места хранения.

OLAP-приложения оперируют с большими массивами данных, уже накопленными в оперативных базах данных OLTP-систем, взятыми из электронных таблиц или из других источников данных. Такие системы характеризуются следующими признаками:

- Добавление в систему новых данных происходит относительно редко крупными блоками (например, раз в квартал загружаются данные по итогам квартальных продаж из OLTP-системы).
- Данные, добавленные в систему, обычно никогда не удаляются и не изменяются.
- Перед загрузкой данные проходят различные процедуры "очистки", связанные с тем, что в одну систему могут поступать данные из многих источников, имеющих различные форматы представления, данные могут быть некорректны, ошибочны.
- Запросы к системе являются нерегламентированными и, как правило, достаточно сложными. Очень часто новый запрос формулируется аналитиком для уточнения результата, полученного в результате предыдущего запроса.
- Скорость выполнения запросов важна, но не критична.

Исходя из перечисленных признаков OLAP-систем, можно сделать вывод, что база данных такой системы может быть в значительной степени денормализованной. Поскольку основным видом запросов к базе данных являются запросы на выборку, положительные моменты нормализации не могут быть использованы, а сокращение операций соединения в запросах окажется весьма полезным.

В последнее время активно развивается еще одно направление аналитической обработки данных, получившее название *Data Mining (осмысление данных, иногда говорят «раскопка данных»)*. Это направление направлено на поиск скрытых закономерностей в данных и решение задач прогнозирования. Приложения Data Mining также не изменяют данные, с которыми они работают, поэтому для них более предпочтительной является денормализованная база данных.

Для того чтобы подчеркнуть особый способ организации данных, которые могут эффективно использоваться для анализа приложениями OLAP и Data Mining, к ним применяют специальный термин *«хранилища данных» (DataWare House)*. Важно отметить, что хранилища данных, в отличие от оперативной БД, хранят исторические данные, т.е. отражают те факты из деятельности предприятия, которые уже произошли, следовательно, могут храниться в неизменном виде («историю не переписывают») и накапливаться годами, в связи с чем их размеры могут стать весьма внушительными. После перекачки данных в хранилище они обычно удаляются из оперативной БД, что позволяет поддерживать ее размеры в заданных пределах.

Таким образом, можно представить корпоративную информационную систему современного предприятия в виде совокупности нескольких

различных подсистем, среди которых есть OLTP, OLAP-системы, а также приложения Data Mining (рис 3.12).

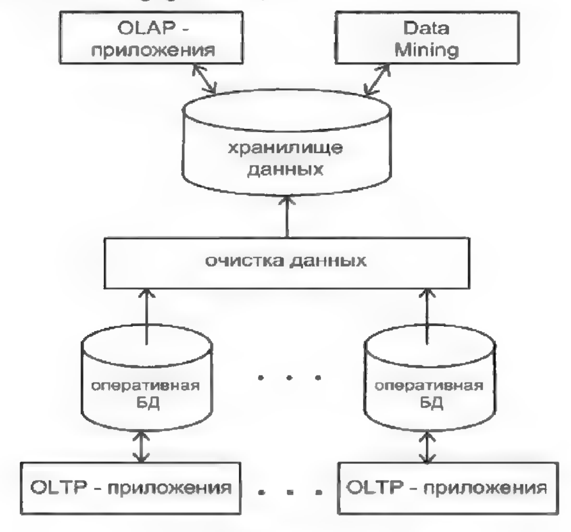


Рис. 3.12. Интегрированная информационная система предприятия

Корпоративные данные могут собираться в одной или нескольких оперативных БД, а все исторические данные концентрируются в едином хранилище, которое объединяет *все данные* в целях выполнения различных видов их анализа и предоставления управленческому персоналу информации, необходимой для принятия обоснованных решений по управлению бизнесом.

3.3.3. Хранилища данных

Ральф Кимбалл (Ralph Kimball), один из авторов концепции хранилищ данных, описывал хранилище данных как «место, где люди могут получить доступ к своим данным». Он же сформулировал и основные требования к хранилищам данных:

- поддержка высокой скорости получения данных из хранилища;
- поддержка внутренней непротиворечивости данных;
- возможность получения и сравнения так называемых срезов данных (slice and dice);
- наличие удобных утилит просмотра данных в хранилище;
- полнота и достоверность хранимых данных;
- поддержка качественного процесса пополнения данных.

Типичная структура хранилищ данных

В отличие от оперативных баз данных, хранилища данных проектируются таким образом, чтобы время выполнения запросов на выборку данных было минимальным. Обычно данные копируются в хранилище из оперативных баз данных согласно определенному расписанию (раз в день, раз в месяц, раз в квартал).

Типичная структура хранилища данных существенно отличается от структуры обычной реляционной БД. Как правило, эта структура денормализована (это позволяет повысить скорость выполнения запросов), поэтому может допускать избыточность данных.

Логически структуру хранилища данных можно представить как многомерную базу данных, которая представляет собой так называемый OLAP-куб. OLAP-куб имеет несколько измерений, которые можно считать осями координат (если таких измерений три, то тогда уместна геометрическая интерпретация в виде куба, на практике обычно бывает более трех измерений, которые не отобразить никакой геометрической фигурой). На пересечении осей располагаются показатели (один или несколько – как получится), они и являются предметом многомерного анализа.

Основной операцией, применяемой к OLAP-кубам, является операция *агрегирования показателей* (т.е. вычисления агрегатных функций, таких как сумма, минимальное, максимальное, среднее значение показателя) применительно к различным измерениям. Например, можно вычислить суммарные объемы продаж за различные периоды времени, по отдельным группам товаров, по различным регионам и т.д.



Рис. 3.13. Типичная структура хранилища данных – схема «звезда»

Реализация OLAP-кубов может быть различной. В последнее время наиболее распространенным вариантом является использование денормализованной реляционной структуры. В этом случае основными составляющими структуры хранилищ данных (рис.3.13) являются таблица фактов (fact table) и таблицы измерений (dimension tables), соединенные по

схеме «звезда» (star schema). Название «звезда» используется в том случае, если каждое измерение содержится в одной таблице размерности.

Таблица фактов

Таблица фактов является основной таблицей хранилища данных. Как правило, она содержит сведения об объектах или событиях, совокупность которых будет в дальнейшем анализироваться. Обычно говорят о четырех наиболее часто встречающихся типах фактов. К ним относятся:

- факты, связанные с транзакциями (Transaction facts). Они основаны на отдельных событиях (типичными примерами которых являются телефонный звонок или снятие денег со счета с помощью банкомата);
- факты, связанные с «моментальными снимками» (Snapshot facts). Основаны на состоянии объекта (например, банковского счета) в определенные моменты времени, например на конец дня или месяца. Типичными примерами таких фактов являются объем продаж за день или дневная выручка;
- факты, связанные с элементами документа (Line-item facts). Основаны на том или ином документе (например, счете за товар или услуги) и содержат подробную информацию об элементах этого документа (например, количестве, цене, проценте скидки);
- факты, связанные с событиями или состоянием объекта (Event or state facts). Представляют возникновение события без подробностей о нем (например, просто факт продажи или факт отсутствия таковой без иных подробностей).

Таблица фактов, как правило, содержит уникальный составной ключ, объединяющий первичные ключи таблиц измерений. Чаще всего это целочисленные значения либо значения типа «дата/время». Таблица фактов может содержать сотни тысяч или даже миллионы записей, и хранить в ней повторяющиеся текстовые описания, как правило, невыгодно – лучше поместить их в меньшие по объему таблицы измерений. При этом как ключевые, так и некоторые неключевые поля должны соответствовать измерениям OLAP-куба. Помимо этого таблица фактов содержит одно или несколько числовых полей для хранения показателей, на основании которых в дальнейшем будут получены агрегатные данные.

Отметим, что для многомерного анализа пригодны таблицы фактов, содержащие как можно более подробные данные (то есть данные, соответствующие самой детальной таблице оперативной БД). Например, в банковской системе в качестве факта можно принять одну транзакцию клиента (снять деньги со счета, положить, перевести на другой счет и т.д.). В системе предприятия, работающего в сфере торговли или услуг, фактом может быть каждая продажа или каждая услуга, оказанная клиенту.

Таблицы измерений

Таблицы измерений содержат неизменяемые либо редко изменяемые данные. Таблицы измерений содержат как минимум одно описательное поле (обычно с именем члена измерения) и, как правило, целочисленное ключевое поле (обычно это суррогатный ключ) для однозначной идентификации члена измерения. Нередко таблицы измерений содержат некоторые дополнительные атрибуты членов измерений, содержавшиеся в исходной оперативной базе данных (например, адреса и телефоны клиентов).

Каждая таблица измерений должна находиться в отношении *один ко многим* с таблицей фактов.

Очень многие измерения могут представлять собой иерархию. Например, если вычислять агрегатные данные продаж по регионам, то можно выделить уровни отдельных населенных пунктов, областей, округов, стран, которые представляют собой иерархию. Для представления иерархии в реляционной БД может использоваться несколько таблиц, связанных отношением *один ко многим* и соответствующих различным уровням иерархии в измерении, или одна таблица с внутренними иерархическими связями.

Если хотя бы одно из измерений содержится в нескольких связанных таблицах, такая схема хранилища данных носит название «снежинка» (snowflake schema). Дополнительные таблицы измерений в такой схеме, обычно соответствующие верхним уровням иерархии измерения и находящиеся в соотношении «один ко многим» с таблицей измерений, соответствующей нижнему уровню иерархии, иногда называют консольными таблицами (outrigger table). Схема «снежинка» изображена на рис. 3.14.

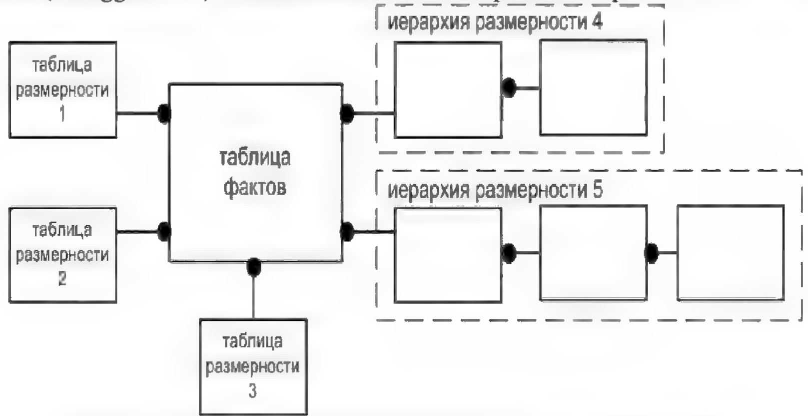


Рис. 3.14. Структура хранилища данных – схема «снежинка»

Если измерение, содержащее иерархию, основано на одной таблице измерений, то она должна содержать столбец, указывающий на «родителя» данного члена в этой иерархии. Например, населенный пункт в качестве родителя имеет область, область – округ, округ – страну, а у страны в качестве родителя обычно устанавливается какое-либо значение по умолчанию, соответствующее отсутствию родителя. Отметим, что скорость роста таблиц измерений должна быть незначительной по сравнению со скоростью роста таблицы фактов; например, добавление новой записи в таблицу измерений, характеризующую товары, производится только при появлении нового товара, не продававшегося ранее.

Одно измерение куба может содержаться как в одной таблице (в том числе и при наличии нескольких уровней иерархии), так и в нескольких связанных таблицах.

Отметим, что даже при наличии иерархических измерений с целью повышения скорости выполнения запросов к хранилищу данных нередко предпочтение отдается схеме «звезда».

Принципы организации хранилища

1. Проблемно-предметная ориентация: данные объединяются в категории и хранятся в соответствии с областями, которые они описывают, а не с приложениями, которые они используют.

2. Интегрированность: объединяет данные т.о., чтобы они удовлетворяли всем требованиям всего предприятия, а не единственной функции бизнеса.

3. Некорректируемость: данные в хранилище данных не создаются, т.е. поступают из внешних источников, не корректируются, не удаляются.

4. Зависимость от времени: данные в хранилище точны и корректны только в том случае, когда они привязаны к некоторому промежутку или моменту времени.

4. Язык SQL

Цель изучения данной главы – получить компетенции использования языка SQL (языков DDL и DML) при разработке и эксплуатации информационных систем.

После изучения главы вы будете:

- знать команды DDL, уметь ими пользоваться
- знать основные объекты реляционной базы данных (таблицы, индексы, представления, хранимые процедуры и функции, триггеры) и использовать их по назначению в процессе разработки приложений баз данных
- уметь грамотно определять типы полей и ограничения целостности в процессе создания и изменения структуры таблиц
- понимать логику разработки запросов к базам данных, свободно использовать команды языка DML
- уметь пользоваться представлениями (создавать, удалять, изменять)
- уметь создавать хранимые процедуры, функции и триггеры, используя одно из процедурных расширений языка SQL (PL/SQL)

Введение в SQL

Первые языки запросов для реляционных баз данных появились в 70-е годы. В то время существовало несколько различных языков запросов, что порождало определенные неудобства у производителей СУБД. Непосредственным предшественником языка SQL явился язык SEQUEL.

Стандарт языка SQL постоянно расширяется, поэтому к настоящему времени имеется несколько официально утвержденных вариантов:

1. SQL-89, первый неполный вариант, практика быстро показала, что он нуждается в расширении.
2. SQL-92 (или SQL-2), значительно расширенная версия, многие СУБД в настоящее время гарантируют полную поддержку SQL-2 и частичную – более поздних стандартов.
3. SQL-99, введены еще некоторые расширения.
4. SQL-2003, самый полный вариант стандарта, в котором учтены многие решения, уже реализованные разработчиками СУБД и ставшие стандартом de-facto (например, стандартизирован объект sequence, который давно используется в некоторых СУБД, например, Oracle, стандартизированы почти все встроенные типы данных, используемые в различных СУБД).

Стандарт SQL-2003 содержит довольно много команд, которые охватывают различные аспекты функционирования ИС и разделены на 7 классов команд. Тем не менее, основу языка SQL составляют два класса

команд, которые принято называть языками: язык определения данных и язык манипулирования данными. В оригинале - DDL (data definition language) и DML (data manipulation language).

В данном учебном модуле рассмотрим только эти два класса (DDL и DML). Остальные классы, которые в стандарте SQL-92 объединены общим названием «язык управления доступом» (data control language – DCL), будут частично рассмотрены в следующем учебном модуле.

При изложении материала будем ориентироваться на стандарт SQL–2003, однако следует иметь в виду, что те команды, которые нам предстоит изучить, практически не изменились со времен SQL-92.

Производители СУБД, в целом поддерживая стандарт SQL, тем не менее, иногда допускают незначительные изменения синтаксиса в отдельных командах. Практически все СУБД содержат те или иные расширения языка SQL. Поэтому изучать язык SQL, не привязываясь ни к какой конкретной СУБД, можно только чисто теоретически. Поскольку целью данного модуля является получение компетенций (т.е. умения применить полученные знания на практике), необходимо остановиться на какой-либо конкретной СУБД, которая будет использоваться для отладки демонстрационных примеров и практических заданий модуля.

В качестве такой СУБД была выбрана самая распространенная на мировом рынке Oracle. В диалекте SQL для СУБД Oracle отклонения от стандарта минимальны. Приведенные в главе примеры отлаживались с использованием клиентской консольной утилиты SQL *Plus на сервере Oracle 10g.

Везде в описании команд, где имеется хотя бы небольшое отклонение синтаксиса Oracle от стандарта SQL 2003, это будет отдельно оговариваться. Учитывая широкое распространение таких СУБД как Microsoft SQL Server, MySQL и ряда других, иногда будут оговариваться особенности и этих СУБД.

Формат записи операторов SQL свободный. Везде, где имеется пробел или знак препинания, может быть вставлено любое число пробелов или переходов на новую строку. При работе в консольной утилите SQL *Plus используется символ ; (точка с запятой) как признак окончания запроса – текст запроса, завершенный точкой с запятой, немедленно отсылается на сервер для выполнения. Однако точка с запятой не является частью команды SQL, поэтому в приводимых в лекциях примерах она будет отсутствовать.

При описании языка мы сочли возможным не приводить полный синтаксис каждой команды в формальной форме (формы Бэкуса-Наура или синтаксические диаграммы), учитывая лаконичность и относительную простоту синтаксиса, а также наличие отклонений от стандарта в диалектах языка и изменчивость самого стандарта и диалектов. Синтак-

сис команд описывается неформально, с пропуском некоторых незначительных, с нашей точки зрения, конструкций, но достаточно строго:

- все ключевые слова языка записываются прописными буквами, а имена, формируемые пользователями, - строчными, хотя, с точки зрения лексики языка SQL, различия в регистре символов несущественны;
- все необязательные элементы команды заключаются в квадратные скобки;
- в тех случаях, где пробел может трактоваться неоднозначно, он заменяется знаком подчеркивания;
- при пропуске части команды используется многоточие.

Возможные недостатки неформального подхода к описанию компенсируются большим количеством примеров и дополнительными словесными комментариями.

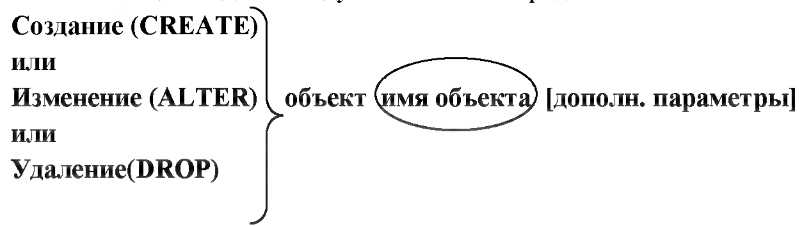
Некоторые локализованные версии СУБД допускают использование в именах, формируемых пользователями, символов национальных алфавитов, но рекомендуем применять эту возможность с чрезвычайной осторожностью. Ключевые символы языка не могут использоваться в качестве имен (таблиц, столбцов, представлений и т.д.) – большая часть подобных ошибок выявляется при компиляции, но в редких случаях ошибки могут быть непредсказуемыми, особенно в части применения символов национальных алфавитов.

4.1. Язык DDL. Основные объекты базы данных

Язык DDL (Data Definition Language) служит для создания, удаления и модификации всех объектов, входящих в состав базы данных.

4.1.1. Общий вид команд DDL

Язык DDL содержит всего три команды **CREATE**, **ALTER** и **DROP**, которые могут быть применены к различным объектам базы данных.
В самом общем виде команду DDL можно определить так:



Примеры:

DROP TABLE t - удаление таблицы с именем t, здесь объектом является TABLE, его имя t.

Данная команда не требует никаких дополнительных параметров CREATE TABLE tt (n NUMBER, x VARCHAR(50)) – создание таблицы с именем tt, в качестве дополнительных параметров задаются определения столбцов, в данном прмере их два: столбец n имеет цифровой тип, а столбец x - текстовый тип, причем длина текста не превышает 50 символов.

4.1.2. Основные объекты БД

Современные базы данных, кроме таблиц с данными, содержат еще ряд объектов, необходимых для осуществления эффективного доступа и обработки данных. Кратко охарактеризуем каждый из объектов.

1. База данных (DATABASE) – контейнер, в котором будут содержаться таблицы и другие объекты, которые мы рассмотрим ниже. Как правило, СУБД может обслуживать одновременно несколько различных баз данных, если в этом есть необходимость. В СУБД Oracle в процессе ее установки создается база данных, которая при работе может быть использована по умолчанию, поэтому во многих случаях нет необходимости создавать другие базы данных.

2. Схема (SCHEMA) – часть базы данных, в пределах которой все имена создаваемых объектов должны быть уникальны. В разных схемах одной и той же базы данных разрешены одинаковые имена, например, таблиц. Схемы поддерживаются далеко не всеми СУБД. СУБД Oracle поддерживает это понятие по-своему. Отдельной команды CREATE SCHEMA … в этой СУБД нет, но при создании нового пользователя ему предоставляется в распоряжение собственная схема, имя которой совпадает с именем пользователя. Если пользователь наделен правами создавать объекты в базе данных, владея схемой, он будет владеть всеми правами на те объекты, которые создаст в своей схеме. Эта тема будет подробно рассматриваться в последнем учебном модуле курса.

3. Таблица (TABLE) – безусловно, основной объект базы данных. В стандарте SQL таблица определяется как мультимножество строк, в отличие от реляционной теории, где отношение (математическая модель таблицы) определяется как множество кортежей (кортеж - математическая модель одной строки). Мультимножество является расширением понятия множества, в котором допускаются повторяющиеся элементы.

Иными словами, стандарт SQL допускает создание таблицы, в которой не определено ни одного потенциального ключа (т.е. могут быть одинаковые строки). Однако на практике принято определять первичный ключ даже для таких таблиц, на которые не ссылаются другие таблицы.

4. Индекс (INDEX) - вспомогательный объект, который служит для ускорения поиска данных, однако замедляет операции вставки, удаления и обновления строк таблиц. В качестве структуры данных при реализации индексов, как правило, используются сильноветвящиеся деревья во внешней памяти (B+- деревья), которые автоматически обновляются при изменении данных таблицы. Кроме древовидных индексов, многие СУБД предоставляют возможность создавать индексы и на основе других структур (хеш-таблицы, битовые карты и др.).

Для всех потенциальных ключей таблицы автоматически создаются древовидные индексы, по остальным столбцам можно создавать индексы при помощи команды CREATE INDEX … . Можно создавать индексы и по нескольким столбцам одновременно, некоторые СУБД позволяют создавать индексы на основе выражений.

Команда CREATE INDEX ... не входит в стандарт SQL 2003, однако поддерживается всеми СУБД без исключения. Наличие или отсутствие индексов сильно влияет на производительность СУБД, поэтому индексы по праву считаются очень важным, хотя и скрытым от пользователя, объектом БД, требующим пристального внимания разработчиков приложений баз данных и АБД.

5. Представление (VIEW) – именованный запрос на выборку, который хранится в БД и выполняется на сервере при любом обращении к нему по имени, создавая при этом виртуальную таблицу с отобранными данными. Представления позволяют предоставлять пользователям любые выборки данных, с которыми можно работать практически так же, как и с физическими таблицами, входящими в состав БД. Иными словами, механизм представлений позволяет конструировать производные виртуальные таблицы на основе базовых таблиц базы данных.

6. Хранимая процедура или функция (STORED PROCEDURE, FUNCTION). Данные объекты БД пишутся на языке процедурного расширения языка SQL, который дополняет язык SQL такими управляющими структурами языка высокого уровня, как ветвления и циклы, и позволяет реализовать любые алгоритмы обработки данных. Хранимый код постоянно хранится на сервере и выполняется по запросу на его запуск из приложений клиентов.

Управляющие конструкции процедурных расширений SQL не регламентируются стандартом, поэтому большинство СУБД имеют свои собственные процедурные расширения. Однако команды CREATE PROCEDURE … и CREATE FUNCTION ... являются частью стандарта, который регламентирует также правила встраивания команд SQL в код хранимой подпрограммы (определяется понятие Embedded SQL – встроенный SQL).

7. Триггер (TRIGGER) – особый вид хранимой процедуры, который срабатывает автоматически при наступлении определенных событий в базе данных. Основными такими событиями являются вставка, удаление и обновление строк, однако некоторые СУБД предоставляют возможность создавать триггеры и на другие события, например, открытие и закрытие сеанса связи с сервером, ряд команд DDL и т.д. Триггеры – это очень мощный (и опасный) инструмент в руках администратора базы данных (о существовании триггера может не знать даже разработчик прикладного ПО, не говоря уж о конечных пользователях).

В качестве дополнительной информации перечислим еще ряд объектов базы данных, которые позволяют СУБД реализовать несколько важных функций управления данными.

8. Последовательность (SEQUENCE) – в некоторых СУБД (Oracle) служат для генерации уникальных значений суррогатных первичных ключей таблиц. Присвоение этих значений ключевому столбцу в Oracle выполняется в триггере на вставку новой строки.

9. Пользователь и роль (USER, ROLE) – пользователи и их права на выполнение различных действий в базе данных. Эти объекты служат для разграничения доступа к информации многочисленным пользователям БД, которые совместно используют общие объекты базы данных и могут выполнять только те действия, которые определяются их ролями.

10. Связь, снимок, синоним (LINK, SNAPSHOT, SYNONYM) данные объекты используются при организации распределенных баз данных, в которых данные хранятся на нескольких серверах (узлах), обычно удаленных друг от друга территориально. Для того, чтобы физически распределенные данные, логически воспринимались пользователями как единая целостная база данных, между серверами устанавливают связи («линки»- объект LINK), а для удобного обращения к удаленным объектам используют короткие синонимы (объект SYNONYM) вместо длинных составных имен.

Снимок (объект *SNAPSHOT*) – таблица или представление, которое посылается на удаленный сервер и периодически обновляется в автоматическом режиме. Снимок, в отличие от представления, это реальная физическая таблица, хранящаяся на удаленном сервере, которая позволяет избежать многочисленных запросов пользователей удаленного сервера к данным, хранящимся на другом сервере. Однако не стоит забывать, что снимок не может обновляться очень часто, поэтому в какие-то моменты времени его данные могут не соответствовать актуальному состоянию удаленной БД.

Любая база данных должна содержать, как минимум, одну таблицу, реальные базы обычно содержат десятки, сотни и даже тысячи таблиц. Все остальные объекты создаются по мере необходимости.

4.2. Команды DDL для работы с таблицами

4.2.1. Создание таблицы

При создании таблицы задается ее уникальное имя и структура, т.е. первоначально создается пустая таблица, которая затем наполняется данными и при помощи команд DML. При определении структуры таблицы задаются не только имена и типы каждого столбца, но и ряд ограничений на вводимые данные, которые будут жестко контролироваться СУБД при заполнении таблиц и редактировании данных. Тем самым пользователи СУБД имеют возможность ограничить множество значений для встроенных типов данных СУБД, добавив дополнительные условия проверки. Таким образом в СУБД реализуется понятие домена как множества допустимых значений столбца.

Следует заметить, что в стандарте SQL 2003 добавлена команда CREATE TYPE, которая позволяет определять пользовательские типы данных, однако для большинства стандартных областей применений баз данных хватает и встроенных типов с дополнительными ограничениями. Общий вид команды для создания таблицы:

CREATE TABLE имя_таблицы
(список_определений_столбцов [,список ограничений_на_таблицу]
)

Элементы списка всегда разделяются запятыми (в дальнейшем это будет считаться само собой разумеющимся и особо оговариваться не будет).

Определение каждого столбца в списке имеет вид:
имя_столбца тип_столбца [DEFAULT значение_по_умолчанию] [ограничения_на_столбец]

Обязательными элементами описания столбца являются имя столбца и тип данных в столбце.

Имя столбца должно быть уникальным в пределах таблицы.

Стандартом SQL предусматривается набор основных типов данных. Внутреннее представление и ограничения на размер типов стандартом не устанавливаются, они по-разному реализуются в разных СУБД. Не все СУБД в полном объеме реализуют стандартные типы данных, кроме того,

различные СУБД могут иметь собственные типы данных. Ниже мы приводим стандартные типы данных SQL 2003 и основные типы данных, поддерживаемые СУБД Oracle.
При определении столбца для него может быть задано значение по умолчанию (DEFAULT). Если при занесении данных в строку таблицы не указывается значение для этого столбца, в столбец заносится значение по умолчанию. Если значение по умолчанию для какого-либо столбца при создании таблицы не задается, в случае отсутствия данных элемент этого столбца получает значение NULL.

Типы данных

Числовые типы

Стандарт предусматривает следующие числовые типы.
SMALLINT, INTEGER (INT) и BIGINT (последний появился только в SQL 2003) - целые двоичные числа со знаком, хотя стандарт не оговаривает размеры, в большинстве случаев это 2, 4 и 8 байт соответственно.
NUMERIC(p,s) или NUMBER(p,s) или DECIMAL(p,s) - типы-синонимы для десятичных чисел с фиксированной точкой. Число p задает общее количество десятичных разрядов в числе, s - число разрядов после точки.
FLOAT(p), REAL, DOUBLE PRECISION - двоичные числа с плавающей точкой. Разрядность типов REAL и DOUBLE PRECISION зависит от реализации.

В Oracle, в отличие от большинства других СУБД, внутреннее представление всех числовых данных одинаково – все числа хранятся в двоично-десятичном представлении (т.е. для каждой десятичной цифры числа хранится ее четырехбитное двоичное представление). Поэтому в Oracle имеется единственный числовой тип.
NUMBER[(p[,s])]

Oracle однако принимает и все стандартные числовые типы и автоматически переводит их во внутреннее представление NUMBER. Например, стандартный тип INTEGER она автоматически приводит к типу NUMBER(38) – число без десятичной точки из 38 десятичных разрядов (один из них – знак числа) и отводит под него 38 байт, что в большинстве случаев избыточно. В целях рационального использования дискового пространства при работе с Oracle разумно пользоваться типом NUMBER. Например, для целых чисел (соответствующих типу INTEGER) разумным типом будет NUMBER(10).

Вообще, для любой СУБД необходимо сначала внимательно изучить документацию по поддерживаемым типам и способам их внутреннего представления, а затем приступать к созданию таблиц. Знание внутренне-

го представления данных поможет не только сэкономить пространство на диске, но и повысить производительность СУБД за счет ускорения операций обмена между внешней и оперативной памятью.

Константы числовых типов записываются так же, как и десятичные константы в большинстве языков программирования. Для действительных констант возможна нотация как с десятичной точкой, так и в E-форме.

Символьные типы

CHAR[ACTER] [(размер)] строка фиксированной длины (по умолчанию размер типа – 1 байт), максимальный размер в Oracle 2000 байт, но во многих СУБД размер этого типа ограничивается 256 байтами.

VARCHAR(размер) (в Oracle дополнительно поддерживается тип VARCHAR2, идентичный VARCHAR, но гарантированно неизменный вне зависимости от возможного изменения типа VARCHAR в стандарте) — строка переменной длины, в Oracle его размер не более 4000 байт.

Данные типов CHAR и VARCHAR хранятся по-разному. Под CHAR всегда отводится столько байт, сколько указано в размере, а под данные типа VARCHAR память на диске выделяется в соответствии с реальными размерами текста (все завершающие текст пробелы автоматически удаляются) плюс четыре байта для хранения этого размера. Пустая строка занимает 4 байта и содержит размер 0. В связи с этим для хранения таких данных, как наименования каких-либо объектов, предпочтительнее тип VARCHAR. Если предполагается, что столбец может содержать много NULL-значений (например, столбец «Примечания» или «Пояснения»), то тип VARCHAR также выигрывает перед типом CHAR.

Стандартом также поддерживаются типы NCHAR (NATIONAL CHARACTER) и NVARCHAR, которые были введены для поддержки символов национальных алфавитов. Оба типа реализованы в Oracle и используются в том случае, если для кодирования символов используется Unicode.

Константы символьного типа берутся в апострофы. Апостроф внутри символьной константы кодируется двумя апострофами.

Типы даты и времени

Стандарт поддерживает типы DATE, TIME и TIMESTAMP, в Oracle реализован тип DATE, которые служит для хранения даты и времени суток, а, начиная с версии 9i, еще и TIMESTAMP, который позволяет хранить момент времени с высокой точностью (до миллисекунд).

Над датами выполняется операция вычитания, при этом разность двух дат вычисляется в днях. К дате можно прибавить или вычесть задан-

ное количество дней, при этом получается другая дата, можно сравнивать две даты, используя обычные знаки сравнения. Внутренний формат представления типа Date в Oracle – действительное число, целая часть которого хранит количество дней, содержащихся в дате, а действительная – время суток.

Константы типа даты заключаются в апострофы, как и текстовые константы, их формат должен соответствовать настройкам сервера.

Типы для хранения больших объектов

Современные базы данных позволяют хранить не только хорошо структурированную фактографическую информацию, но и такие данные, как рисунки (фотографии), звуковые или анимационные файлы, тексты формата Microsoft Word, Web-страницы в формате html и другую слабо структурированную информацию. Для хранения подобных данных используются типы больших объектов.

Стандарт поддерживает типы BLOB (Binary Large Object) и CLOB (Character Large Object) или NCLOB (National Character Large Object).

Oracle дополнительно поддерживает тип LONG для хранения больших текстов. Работа с типами LONG и CLOB выполняется значительно медленнее, чем с типом VARCHAR, поэтому они используются только для действительно больших текстов.

Следует заметить, что, начиная с версии 8, в Oracle появилось специальное расширение Oracle Text, предназначенное для эффективной работы с большими текстами.

Ограничения

Ограничения (constraints) позволяют задавать дополнительные условия проверки вводимых данных, которые СУБД проверяет автоматически. Данные, которые не удовлетворяют условиям, заданным в ограничениях, отвергаются. Например, при вставке новой строки в таблицу она не будет добавлена, если хотя бы одно из значений не удовлетворяет ограничениям. Механизм ограничений позволяет поддерживать данные в непротиворечивом состоянии, соответствующем бизнес-правилам предметной области.

Любое ограничение может быть поименовано, в противном случае, имя для ограничения СУБД создает автоматически. Для явного задания имени, к описанию ограничения следует добавить слева фразу CONSTRAINT имя_ограничения (рекомендуется задавать явные имена для ограничений).

Можно задать ограничения для отдельного столбца или для таблицы в целом.

Ограничения столбца:

NOT NULL/NULL – допустимы ли пустые (неопределенные) значения в столбце, по умолчанию используется значение NULL (т.е. допустимы), что в большинстве случаев не соответствует бизнес-правилам предметной области, поскольку пропуски какой-либо нужной информации обычно недопустимы. Данное ограничение не именуется.
PRIMARY KEY – ограничение первичного ключа, при этом автоматически накладывается ограничение NOT NULL. При задании значений первичного ключа любое значение проверяется на уникальность и при обнаружении дубликата операция прерывается. В таблице может быть только один столбец с ограничением PRIMARY KEY.
UNIQUE – ограничение уникальности (альтернативный ключ), ограничение NOT NULL также накладывается автоматически. Фактически, UNIQUE ничем не отличается от PRIMARY KEY, однако количество столбцов с UNIQUE не ограничено.
REFERENCES имя_главной_таблицы — внешний ключ, который задается для столбца подчиненной (детальной) таблицы. Для значений внешнего ключа автоматически выполняется проверка на существование равного значения первичного ключа главной таблицы. При определении внешнего ключа могут быть дополнительно определены правила обеспечения ссылочной целостности. Например, правилами для удаления являются:

- ON DELETE RESTRICT – запретить удаление строки главной таблицы, если в подчиненной есть хотя бы одна строка, которая на нее ссылается;
- ON DELETE CASCADE – вместе со строкой главной таблицы автоматически удалить все ссылающиеся на нее строки подчиненной таблицы;
- ON DELETE SET NULL – при удалении строки главной таблицы установить во внешнем ключе ссылающихся строк значение NULL (это правило может быть определено только в том случае, если на внешний ключ не наложено ограничение NOT NULL);
- ON DELETE SET DEFAULT – при удалении строки главной таблицы установить во внешнем ключе значение по умолчанию (это правило может быть определено только в том случае, если при определении внешнего ключа задано DEFAULT значение_по_умолчанию).

По умолчанию принят запрет удаления строк при наличии ссылок на них (ON DELETE RESTRICT).

Аналогичные правила можно определить для обновления (ON UPDATE …), однако необходимости в них обычно не возникает, т.к. первичные ключи обновлять не принято.

CHECK – проверка логических выражений при изменении данных в столбце, например, CHECK (имя_столбца BETWEEN 1 AND 100).
Например:

```
CREATE TABLE t
(c1 NUMBER(8) DEFAULT 0 NOT NULL CONSTRAINT un UNIQUE,
 c2 VARCHAR(100) NOT NULL
)
```

Создается таблица с именем t, содержащая два столбца: числовой столбец c1, обязательный для заполнения и принимающий значение 0 по умолчанию, все значения этого столбца уникальны (ограничение уникальности имеет имя un) и текстовый столбец c2, обязательный для заполнения текстом, его размер не более 100 символов.

Ограничения таблицы

Данные ограничения используются в том случае, если они затрагивают сразу несколько столбцов:
PRIMARY KEY (список столбцов) — составной первичный ключ;
UNIQUE (список столбцов) — составной альтернативный ключ;
FOREIGN KEY имя_внешнего_ключа (список столбцов) REFERENCES имя_главной_таблицы [правила поддержки ссылочной целостности];
CHECK (логическое выражение, затрагивающее сразу несколько столбцов).
Например:

```
CREATE TABLE t1
( c1 NUMBER(3) NOT NULL,
 c2 DATE NOT NULL,
 c3 NUMBER(3) NOT NULL,
 CONSTRAINT pk_t1 PRIMARY KEY(c1,c2),
 CONSTRAINT ck_t1 CHECK(c1+c3<=200)
)
```

Создается таблица с именем t1, содержащая три столбца, обязательных для заполнения. Составной ключ таблицы включает столбцы c1 и c2 (обратим внимание, что каждый из столбцов таблицы не обладает свойствами уникальности, поэтому PRIMARY KEY не может быть ограничением одного столбца). Ограничение CHECK также затрагивает сразу два столбца, поэтому оформлено как ограничение таблицы.

4.2.2. Удаление таблиц и изменение их структуры

Команда удаления таблицы имеет вид:
DROP имя_таблицы

Нельзя удалить таблицу, если существуют внешние ключи, ссылающиеся на эту таблицу. Вместе с таблицей удаляются и созданные для нее индексы и триггеры.

Команда изменения структуры таблицы выглядит так:
ALTER TABLE имя_таблицы указания_по_изменению_структуры

Следует подчеркнуть, что команда ALTER TABLE служит для изменения в определении таблицы и может быть применена только к существующим таблицам. В качестве указаний по изменению структуры таблицы могут использоваться следующие фразы:
ADD [COLUMN] определение_столбца – добавить столбец.
Например:
ALTER TABLE t ADD n NUMBER(4) NOT NULL (t— имя существующей таблицы, n — имя нового столбца)

ADD [CONSTRAINT] ограничение — добавить ограничение.
Например:
ALTER TABLE t ADD PRIMARY KEY(n)
или
ALTER TABLE t ADD CONSTRAINT pk_t PRIMARY KEY(n)
В последнем примере ограничение первичного ключа получит имя pk_t

DROP COLUMN имя_столбца
DROP [CONSTRAINT] ограничение
Например:
ALTER TABLE t DROP PRIMARY KEY(n)
или
ALTER TABLE t DROP CONSTRAINT pk_t
ALTER TABLE t DROP COLUMN n
ALTER (MODIFY в Oracle) новое_определение_существующего_столбца

При этом можно изменить тип, размер, ограничение NULL/NOT NULL, значение по умолчанию. Конечно, нельзя изменить имя столбца. Для этого следует удалить столбец со старым именем и добавить новый столбец с нужным именем.

Следует отметить, что во многих случаях команда ALTER TABLE позволяет внести изменения в структуру таблицы, уже заполненной данными (если они удовлетворяют вводимым ограничениям). В некоторых случаях требуется, чтобы столбец был пустым, например, сервер Oracle при удалении столбца требует выполнения этого условия.

Например:
ALTER TABLE t
MODIFY c2 VARCHAR(200)
Увеличили предельный размер текста для столбца c2

4.2.3. Пример создания базы данных

Создадим демонстрационную базу данных из трех таблиц (Студенты, Предметы и Оценки), которая будет хранить сведения об успеваемости студентов. Структура данной БД, конечно, сильно упрощена по сравнению с тем, что требуется для решения реальной задачи учета успеваемости студентов. Однако как пример для обучения основам SQL такая структура вполне подойдет. Назначение каждого столбца на изображенной ниже схеме, очевидно, понятно (рис. 4.1).

Схема Базы Данных:

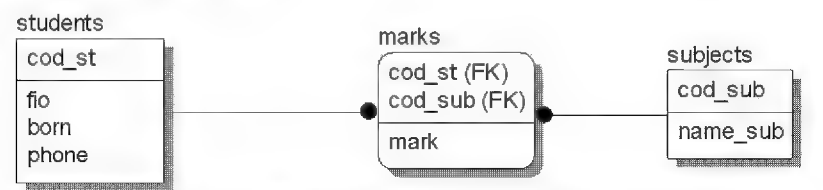


Рис. 4.1. Схема демонстрационной базы данных

Приведем различные варианты команд DML для создания трех таблиц базы данных.

Создание таблицы students:

```
CREATE TABLE students
(
cod_st NUMBER(5) PRIMARY KEY,
name_st VARCHAR(100) NOT NULL,
born DATE NOT NULL,
phone CHAR (15)
)
```

Создание таблицы subjects:

```
CREATE TABLE subjects
(
cod_sub NUMBER (4),
name_sub VARCHAR(200) NOT NULL
)
```

Изменение таблицы subjects, добавление первичного ключа:

```
ALTER TABLE subjects
```

```
ADD PRIMARY KEY (cod_sub)
```

Изменение таблицы subjects, добавление ограничения уникальности названия предмета:

```
ALTER TABLE subjects
ADD UNIQUE (name_sub)
```

Создание таблицы marks:

```
CREATE TABLE marks
(
cod_st NUMBER(5) NOT NULL REFERENCES students
ON DELETE CASCADE,
cod_sub NUMBER(4) NOT NULL REFERENCES subjects ,
mark NUMBER(1) CHECK (mark BETWEEN 2 AND 5),
PRIMARY KEY(cod_st, cod_sub)
)
```

4.2.4. Создание таблиц на основе других таблиц

Можно создать новую таблицу как результат запроса на выборку из одной или более существующих таблиц при помощи команды:

```
CREATE TABLE имя_таблицы AS запрос_на_выборку
```

Например, можно создать новую таблицу, являющуюся копией таблицы students:

```
CREATE TABLE copy_students AS
SELECT * FROM students
```

Новая таблица будет иметь такие же имена и типы столбцов, как и таблица students, содержать точную копию всех данных этой таблицы, но не будет иметь ее ограничений.

4.3. Команды манипулирования данными

Команды CREATE TABLE … и ALTER TABLE …никак не влияют на наполнение таблиц данными. Для этих целей используется язык DML (Data Manipulation Language), который позволяет полностью контролировать процессы наполнения таблиц и измения данных.

Основные команды манипулирования данными:

INSERT – добавить данные
DELETE – удалить данные   изменяют состояние базы данных
UPDATE – изменить данные
SELECT – выбрать данные   без изменения состояния базы данных

Для фиксации изменений, произведенных командами UPDATE, DELETE, INSERT, необходимо выполнить команду фиксации транзакции COMMIT; можно отменить их действие командой ROLLBACK (откат). Подробнее о транзакциях будет рассказано в следующем разделе.

Следует отметить, что при выполнении команд UPDATE, DELETE, INSERT СУБД автоматически проверяет все ограничения, которые были указаны при создании таблиц. Эти проверки, безусловно, существенно замедляют выполнение данных команд. Дополнительное время и другие ресурсы затрачиваются на обеспечение возможности отмены команды (отката). Однако все эти затраты окупаются гарантией того, что правила целостности данных, заложенные в команды DDL, ни при каких обстоятельствах не могут быть нарушены.

4.3.1. Команда INSERT

Вариант 1 – вставка одной строки

Команда имеет вид:

INSERT INTO имя_таблицы [(список_имен_столбцов)]
VALUES (список значений)

Если в команде список имен столбцов опущен, то будут последовательно заполнены все столбцы в том порядке, в котором они были указаны в команде CREATE TABLE.

Например:

INSERT INTO subjects VALUES (5, 'Базы данных')

INSERT INTO students (cod_st, name_st, born) VALUES (3, 'Петров','12.03.1990')

Во второй команде столбец phone (телефон) таблицы students можно оставить незаполненным, т.к. на него не наложено ограничение NOT NULL.

Рассмотрим, при каких ситуациях данные команды могут привести к ошибке. В случае вставки нового предмета таких ситуаций две – уже есть предмет либо с кодом 5, либо с названием Базы данных (оба столбца предполагают проверку уникальности). Во втором случае на уникальность будет проверен только один столбец cod_st.

Команда

INSERT INTO subjects(cod_st, cod_sub, mark) VALUES (3, 5, 1)

в нашей демонстрационной базе данных не имеет никаких шансов выполниться, поскольку на столбец mark наложено ограничение

CHECK (mark BETWEEN 2 AND 5)

Исправим оценку:

INSERT INTO subjects(cod_st, cod_sub, mark) VALUES (3, 5, 4)

Мы уже знаем, что предмет с кодом 5 существует, как и студент с кодом 3, значит, ограничения внешнего ключа не нарушены. Однако может случиться последняя из возможных исключительных ситуаций – нарушение уникальности составного первичного ключа таблицы mark (студент с кодом 3 уже имеет оценку по предмету с кодом 5). В случае, если уникальность не нарушена, новая строка благополучно добавится в таблицу marks.

Вариант 2 – вставка множества строк на основе запроса по другим таблицам

Команда имет вид:

INSERT INTO имя таблицы [(список столбцов)]
SELECT … (запрос на выборку из других таблиц)

Например:

```
INSERT INTO marks (cod_st, cod_sub)
SELECT students.cod_st, subject.cod_sub
FROM students, subject WHERE cod_sub=5
```

В данном примере в таблицу оценок будут добавлены все студенты, а в качестве предмета будет помещен предмет с кодом 5 (команда SELECT будет подробно рассмотрена в следующих лекциях). Теперь преподавателю этого предмета достаточно только проставить оценки в уже созданных строках.

При выполнении команды ограничения проверяются для каждой вставляемой строки и в случае их нарушения хотя бы для одной из строк *все данные не будут добавлены.* Например, если кто-либо из студентов уже имеет оценку по предмету с кодом 5, автоматически выполнится команда ROLLBACK - откат.

Команда не выполнилась бы и в том случае, если бы на столбец mark было наложено ограничение NOT NULL.

4.3.2. Команда DELETE

Общий вид команды:

DELETE FROM имя_таблицы
[WHERE условие_отбора_строк_для_удаления]

Например:

Удалить всех студентов с фамилией Иванов

```
DELETE FROM students
WHERE name_st='Иванов'
```

Более подробно фраза WHERE будет рассмотрена в следуюшем разделе при изучении команды выборки.

Полностью очистить таблицу students

DELETE FROM students

Нельзя удалить строку родительской таблицы, если есть связанные строки в подчиненной таблице и действует правило обеспечения ссылочной целостности ON DELETE RESTRICT. В приведенных выше примерах при определении внешнего ключа cod_st в таблице marks (оценки) было определено правило ON DELETE CASCADE, поэтому вместе со студентами будут каскадом удалены и все их оценки. Последняя команда, таким образом, очистит сразу две таблицы – students и marks.

Зато команда

DELETE FROM subjects

не удалит ничего, если в таблице marks имеется хотя бы одна строка, поскольку при возникновении ошибки будут автоматически отклонены *все изменения*, сделанные этой командой. Если на момент выполнения этой команды таблица оценок была пуста, таблица предметов будет благополучно очищена.

Надо заметить, что для полной очистки таблицы существует более эффективная команда

TRUNCATE TABLE имя_таблицы

Например:

TRUNCATE TABLE marks

4.3.3. Команда UPDATE

Данная команда не изменяет количество строк в таблице, поскольку ее назначение состоит в изменении данных, которые уже имеются в таблице.

Общий вид команды:

UPDATE имя_таблицы SET столбец1=значение [, столбец2=значение …]
[WHERE условие_отбора_строк]

Например:

Прибавим всем студентам по одному баллу к каждой оценке

UPDATE marks SET mark= mark+1

При выполнении команды UPDATE автоматически проверяются все ограничения столбца. В приведенном примере столбец mark имеет ограничение CHECK, которое задает диапазон оценок от 2 до 5. Следовательно, если в таблице имеется хотя бы одно значение оценки, равное 5, при увеличении на единицу оно выйдет за пределы диапазона, а все действия, выполненные командой, будут отменены.

Чтобы гарантированно увеличить все оценки, кроме пятерок, изменим команду:

UPDATE marks SET mark= mark+1
WHERE mark<5

4.4. Команда выборки данных (SELECT)

Команда выборки SELECT является одной из самых мощных команд языка SQL, которая реализует все операции реляционной алгебры и некоторые предложения реляционного исчисления. Результатом команды выборки всегда является новая таблица (возможно, пустая или содержащая одно значение, которое трактуется как таблица из одной строки и одного столбца).

Новая таблица является виртуальной, т.е. она существует (обычно в оперативной памяти) до тех пор, пока в ней есть необходимость, она может участвовать в другой команде SQL (CREATE для различных объектов БД, INSERT), в другой команде SELECT (а также DELETE или UPDATE) как результат вложенного запроса или передается клиенту для последующей обработки.

Напомним, что при выполнении любой, сколь угодно сложной, команды выборки состояние базы данных не изменяется, поскольку в процессе ее исполнения выполняются только операции чтения и обработки данных.

Синтаксис команды, с учетом всех ее нюансов, является громоздким, поэтому приведем лишь общий порядок следования основных предложений (фраз):

SELECT [DISTINCT] список_выражений(* - все столбцы таблицы)
FROM список_таблиц, представлений, запросов
(возможно, с операцией соединения JOIN)
[WHERE условия_отбора_строк]
[GROUP BY список_выражений_для_группировки]
[HAVING условия_отбора_групп_с_агрегатными_функциями]
[ORDER BY список_выражений_для_сортировки]

Поскольку команда SELECT является многофункциональной, разберем по порядку возможности ее практического применения.

4.4.1. Запросы на выборку по одной таблице

Выборка всех данных таблицы

SELECT * FROM имя_таблицы (представления)

В дальнейшем при описании синтаксиса команд во фразе FROM будем употреблять только имя_таблицы, хотя везде вместо имени таблицы может быть имя_представления. С представлениями мы познакомимся в следующем разделе, сейчас достаточно знать, что представление – это виртуальная таблица.

Например: выбрать все данные из таблицы students
SELECT * FROM students

В такой форме запросы ставятся крайне редко, поскольку пересылку на компьютер клиента всех данных таблицы, хранимой на сервере, разумно только для совсем небольших таблиц. В подавляющем большинстве приложений требуется отобрать только часть столбцов и/или строк таблицы. Однако в процессе интерактивной работы, например, с утилитой SQL *PLUS в Oracle, символ * может оказать хорошую услугу, если имена столбцов неизвестны или хочется сократить время набора текста запроса.

Отбор столбцов (операция проекции)
SELECT список_имен_столбцов FROM имя_таблицы

Например:
выбрать столбцы с фамилией и датой рождения из таблицы студентов
SELECT name_st, born FROM students
выбрать только столбец с фамилией из таблицы студентов
SELECT name_st FROM students

Результаты выборки по этим запросам могут не удовлетворить пользователей по двум причинам.

- Результаты не отсортированы и могут следовать в различном порядке в разные моменты времени (вспомним, что результат – таблица, которая по определению представляет собой мультимножество строк, а в множестве или мультимножестве порядок следования элементов является несущественным).
- В результатах могут содержаться строки-дубликаты (особенно во втором запросе при наличии однофамильцев), повторение одной и той же строки во многих случаях только затрудняет восприятие результата.

Устранить перечисленные недостатки совсем несложно.

Удаление строк-дубликатов из результатов запроса

Для этих целей в команду выборки введено ключевое слово DISTINCT, которое записывается сразу после SELECT.

Например, для того, чтобы удалить дубликаты однофамильцев из предыдущего запроса, достаточно немного изменить его текст:
SELECT DISTINCT name_st FROM students

При большом ожидаемом количестве строк-дубликатов в результатах запроса использование DISTINCT позволяет существенно сократить размер выборки.

Например, команда
SELECT DISTINCT mark FROM marks

при существующей пятибальной системе возвратит максимум четыре строки, содержащих значения - 2,3,4,5. Порядок следования этих значений предсказать невозможно, поэтому при необходимости получить результаты запроса в отсортированном виде необходимо явно указать в его тексте параметры, необходимые для сортировки.

Сортировка результатов запроса

Фраза ORDER BY…, необходимая для представления результатов выборки в отсортированном виде, обязательно должна быть последней в команде SELECT, поскольку сортировка результатов всегда завершает выполнение запроса на выборку.

После ключевых слов ORDER BY следует указание, как именно требуется отсортировать результаты. Сортировка может быть выполнена по одному или нескольким столбцам. В последнем случае сначала выполняется сортировка по первому из столбцов, указанных в списке, в случае наличия одинаковых значений, строки с одинаковыми значениями сортируются по второму из столбцов и т.д.

Сортировку можно выполнять по столбцам различных типов, исключая типы больших объектов (BLOB, CLOB). Сортировка по текстовым столбцам выполняется в лексикографическом (алфавитном) порядке, сортировка по столбцам даты и времени – в хронологическом порядке.

По умолчанию значения сортируются по возрастанию (точнее, по неубыванию), но можно установить порядок сортировки «по убыванию» с помощью ключевого слова DESC[ENDING]. Можно и явно указать сортировку по возрастанию с помощью ключеого слова ASC[ENDING], но это никак не отразится на результате. Если сортировка выполняется сразу по нескольким столбцам, DESC необходимо указывать для каждого столбца в списке, для которого необходима сортировка по убыванию.

Например, запрос

SELECT name_st FROM students ORDER BY name_st

возвратит список студентов в алфавитном порядке, а запрос

SELECT name_st FROM students ORDER BY name_st, born

не только отсортирует список в алфавитном порядке, но и расставит однофамильцев в соответствии с их возрастом (первым будет самый старший, т.к. у него самая меньшая дата рождения). Если мы хотим расставить однофамильцев в порядке убывания даты рождения, следует изменить текст так:

SELECT name_st FROM students ORDER BY name_st, born DESC

Если и сортировать фамилии требуется в порядке, обратном алфавитному, текст будет выглядеть так:

SELECT name_st FROM students ORDER BY name_st DESC, born DESC

Из приведенных примеров видно, что сортировать результаты можно не только по столбцам, которые отбираются в запросе, но и по любым другим столбцам таблицы, на которой основан запрос. Если столбец сортировки уже указан в списке отбирающихся в запросе столбцов, во фразе ORDER BY можно указать его порядковый номер в списке. Например, запрос

SELECT name_st, born FROM students ORDER BY 1

отсортирует результаты по первому столбцу в списке (это столбец name_st). Такая дополнительная возможность позволяет немного сократить текст запроса, но никак не влияет на его результаты.

В списке ORDER BY в общем случае можно указывать произвольные выражения, в этом случае сортировка будет выполняться по значениям данного выражения.

Отбор строк (операция выборки)

Все разобранные выше примеры возвращали данные для каждой строки таблицы students (или marks). Такие большие выборки обычно требуются только при формировании отчетов, большинство других задач требует предварительного отбора строк таблицы, необходимых пользователю. Напомним, что строки в таблицах не нумеруются, поэтому такие операции, как «извлечь первую строку таблицы», стандартом не поддерживаются.

В качестве критерия для отбора строк таблицы можно задать любое логическое выражение, используя фразу WHERE. В процессе выполнения запроса быдет выполнен отбор только тех строк, для которых данное выражение принимает значение «истина». Таким образом, результатом операции выборки может быть произвольное число строк таблицы (от нуля до всех строк).

Например, запрос:

SELECT * FROM students WHERE cod_st=10

возвратит все данные о студенте, у которого личный код=10. Такой запрос гарантированно вернет не более одной строки (если студента с кодом 10 в таблице нет, то запрос не вернет ни одной строки, но вернуть, например, две строки он не сможет, т.к. личный код студента обладает свойством уникальности).

Условие отбора строк может включать следующие операции:

- операции сравнения < , > , <= , >= , !=(<>) , =
- операция проверки на отсутствие данных IS NULL
  или их наличие IS NOT NULL
- логические операции: AND, OR, NOT
- операция вхождения значения в заданный диапазон значений
  BETWEEN начальное_значение AND конечное_значение

- операция принадлежности значения заданному множеству
  IN (множество)
- операция соответствия заданному шаблону (для текстовых столбцов)
  LIKE ‘шаблон’

В шаблоне для операции LIKE разрешено использовать два символа-заменителя:

- % заменяет последовательность из любого количества любых символов (в том числе и пустая последовательность)
- _ заменяет один любой символ в заданной позиции.

Например, пусть требуется выбрать фамилии и телефоны всех студентов, фамилия которых начинается на букву «С». Список отсортировать по фамилии. Текст запроса:

```
SELECT namt_st, phone
FROM students
WHERE name_st LIKE ‘C%’
ORDER BY 1
```

Если в приведенном запросе заменить шаблон на ‘%С%’, то такой запрос выберет всех студентов, у которых в фамилии есть хотя бы одна буква «С». Если шаблон заменить на ‘С_____’ (после буквы С стоит пять знаков подчеркивания), то будут выбраны студенты, фамилия которых начинается на букву С и содержит 6 символов.

При поиске по текстовым столбцам, в том числе по шаблону, обычно тонким моментом является чувствительность букв к регистру. Во многих СУБД заглавные и строчные буквы при поиске и сортировке не различаются, но, как правило, имеется возможность настройки чувствительности к регистру. В СУБД Oracle строчные и заглавные буквы в текстовых константах и шаблонах различаются, при сортировке также учитывается регистр букв.

Поэтому, например, при поиске студентов, в фамилии которых есть хотя бы одна буква С, в качестве условия отбора следует использовать

```
WHERE name_st LIKE ‘%C%’ OR name_st LIKE ‘%c%’
```

Следует помнить, что запросы с операцией LIKE обычно выполняются довольно медленно, особенно если символ % стоит в начале шаблона (к такому запросу невозможно подключить древовидный индекс для ускорения поиска). Однако при поиске в текстовых столбцах данная операция используется довольно часто. В Oracle версии 10 появилось и еще более мощное средство – поиск по регулярным выражениям, которое пока в стандарт языка SQL не вошло.

Приведем еще ряд примеров запросов на выборку строк, демонстрирующих применение различных операций.

Пусть требуется выбрать все строки из таблицы оценок, в которых присутствуют оценки 3 или 4. Запрос простой, но может быть выполнен несколькими разными способами:

```
SELECT * FROM marks WHERE mark=3 OR mark=4
SELECT * FROM marks WHERE mark>2 AND mark<5
SELECT * FROM marks WHERE mark IN (3, 4)
SELECT * FROM marks WHERE mark  BETWEEN 3 AND 4
```

Можно привести и еще несколько вариантов записи того же самого запроса, но и приведенных вполне достаточно, чтобы понять синтаксис различных операций.

Требуется выбрать такие строки из таблицы студентов, в которых отсутствует телефон студента.

```
SELECT * FROM students WHERE phone IS NULL
```

Обратим внимание, что запрос

```
SELECT * FROM students WHERE phone = NULL
```

возвратит неправильный ответ, но при этом не будет получено сообщения об ошибке.

Создание вычисляемых столбцов (операция расширения)

Вычисляемый столбец – это фиктивный столбец, данные которого не хранятся в базе данных, а вычисляются на основе данных других столбцов. Вычисляемому столбцу в запросе, как правило, присваивается какое-либо имя – псевдоним. В тексте запроса вычисляемый столбец выглядит так:

SELECT … выражение_для_вычисления [AS] псевдоним …

Вообще, псевдоним можно присвоить и любому столбцу, указанному в списке для отбора, но для вычисляемых столбцов это особенно актуально, поскольку в случае отсутствия псевдонима СУБД формирует имя вычисляемого столбца автоматически, обычно такие имена бывают длинными и неудобочитаемыми.

Например, пусть требуется выводить возраст студента вместо его даты рождения. Для этого необходимо создать вычисляемый столбец на основе выражения, которое вычисляет возраст, используя дату рождения и текущую дату. Приводимый запрос будет работать только в СУБД Oracle, поскольку использует ее функцию sysdate для получения текущей даты. Разность двух дат исчисляется в днях, которые затем переводятся в возраст в годах:

```
SELECT cod_st, name_st, trunc((sysdate-born)/365.25) age FROM students
```

В выражениях для вычисляемых столбцов могут использоваться:

- имена столбцов
- константы
- операции

- функции
- круглые скобки.

Числовые и текстовые константы записываются обычным способом, например, запрос:

SELECT ‘студент’ FROM students

столько раз выведет слово студент, сколько строк имеется в таблице students (разумеется, данный запрос не имеет никакого практического смысла). Можно заметить, кстати, что в Oracle фраза FROM в запросах является обязательной, поэтому в любой базе данных автоматически создается таблица с именем dual, состоящая из одной строки и одного столбца, которая может использоваться в запросах, выводящих константы или системные функции. Например, запрос:

SELECT user FROM dual

возвратит имя пользователя, который открыл тот сеанс связи с сервером, в котором выполняется данный запрос.

Набор арифметических операций является стандартным (в стандарте имеется еще операция возведения в степень **, которая в большинстве СУБД не реализована). Для текстовых данных часто используется операция конкатенации, которая в стандарте и большинстве СУБД обозначается двумя вертикальными линиями (||) вместо привычного знака «+», который также используется в некоторых СУБД (Access, Microsoft SQL Server).

Например, выведем фамилию и телефон студентов в виде одного текстового столбца для тех студентов, телефон которых известен:

SELECT name_st ||’ тел. ‘|| phone AS name_phone FROM students
WHERE phone IS NOT NULL

В выражениях для вычисляемых столбцов могут использоваться также скалярные функции. Стандарт SQL 2003 содержит довольно большой набор функций, которые делятся на скалярные и агрегатные. Агрегатные функции будут подробно рассмотрены далее. Сейчас кратко коснемся скалярных функций. Основная проблема их использования в SQL-запросах состоит в том, что, к сожалению, различные СУБД поддерживают различный набор функций, который часто не совпадает со стандартным набором, определенным в SQL 2003. Поэтому запросы, в которых содержится обращение к функциям, при переносе их на другую СУБД, скорее всего, потребуют изменения текста, что несколько ограничивает применение функций.

Все скалярные функции можно разбить на следующие группы (в скобках приводятся примеры функций, которые поддерживаются в Oracle):

- текстовые функции; используются для управления текстовыми строками, например, для удаления ведущих и завершающих пробелов

(функции LTRIM RTRIM), получения фрагмента строки (функция SUBSTR) или преобразования букв в верхний или нижний регистр (функции UPPER и LOWER)

- числовые функции; используются для выполнения математических операций над числовыми данными, например, для вычисления абсолютных значений (функция ABS) и выполнения алгебраических вычислений (тригонометрические функции, логарифмы, остаток от деления – функция MOD, округление и усечение – функции ROUND и TRUNC)
- функции даты и времени; используются для управления значениями даты и времени и для выборки отдельных частей этих значений, например, для возвращения года, месяца, дня или дня недели для заданной даты
- функции преобразования типов данных, например, преобразовать дату или число в строку текста и наоборот (в стандарте CHAR(x), DATE(x) и т.д., в Oracle аналогичные функции TO_CHAR(x, строка_формата), TO_DATE(x, строка_формата) и т.д.)
- системные функции, например, уже упоминавшиеся имя пользователя (функция USER) или системная дата (функция SYSDATE).

Использование оператора CASE в вычисляемых столбцах

Данный оператор позволяет организовывать ветвления в вычисляемых столбцах, что позволяет внести в запрос элементы процедурной логики. Он аналогичен, например, оператору case из языка Pascal или оператору switch из языка C. Например, запрос:

```
SELECT cod_st, cod_sub, CASE mark
WHEN 5 THEN 'ОТЛИЧНО'
WHEN 4 THEN 'ХОРОШО'
WHEN 3 THEN 'УДОВЛЕТВОРИТЕЛЬНО'
WHEN 2 THEN 'ПЛОХО'
END mark FROM marks
```

вместо цифры выведет оценку в виде соответствующего слова.

Использование агрегатных функций

Важной функцией любой информационной системы является автоматическое формирование отчетности на основе данных, хранящихся в БД. Во многих отчетах требуется отображать не сами данные (их слишком много), а результаты их статистической обработки, например, суммарные или средние значения по различным показателям. Выполнять статистическую обработку данных в клиентских приложениях неэффектив-

но, поскольку при этом придется пересылать по сети большое количество необработанной информации с сервера. Более разумным решением является обеспечение возможности выполнять статистическую обработку данных непосредственно на сервере.

С этой целью в команду SELECT введены агрегатные (статистические, итоговые) функции. Основная особенность этих функций состоит в том, что каждая из них вычисляет одно итоговое значение по какому-либо столбцу (это может быть и вычисляемый столбец) для множества строк таблицы.

В стандарте определено 5 агрегатных функций:
SUM(имя_столбца) – сумма значений заданного столбца,
AVG(имя_столбца) – среднее значение
MIN(имя_столбца), MAX(имя_столбца)– минимальное и максимальное значение
COUNT([DISTINCT] * или имя_столбца) – подсчет количества строк.

Первые две функции (сумма и среднее) могут быть вычислены только по числовым столбцам, максимальное и минимальное значения могут быть определены для столбцов всех типов (кроме больших объектов), при этом строки текста сравниваются в лексикографическом, а даты – в хронологическом порядке.

Например:

Подсчитать средний балл по всем студентам и предметам
SELECT AVG (mark) avg_mark FROM marks

Найти минимальную и максимальную даты рождения студентов
SELECT MIN(born) min_date, MAX(born) max_date FROM students

Функция COUNT, казалось бы, вообще не должна содержать аргументов, поскольку ее назначение – подсчет количества строк, что она и делает, если в качестве аргумента используется символ-заменитель *. Однако, если в качестве аргумента использовать имя столбца, данная функция будет подсчитывать количество непустых значений в данном столбце. Второй вариант аргумента разумно использовать, только если столбец не имеет ограничения NOT NULL. Использование ключевого слова DISTINCT приводит к тому, что подсчитывается количество уникальных значений в заданном столбце.

Например:

Посчитать количество студентов (количество строк в таблице students)
SELECT COUNT (*) count_students FROM students

Подсчитать количество студентов, для которых известен номер их телефона.
SELECT COUNT (phone) count_phones FROM students

Подсчитать количество уникальных значений оценок в таблице marks:
SELECT COUNT (DISTINCT mark) count_marks FROM marks

Отметим, что запросы, вычисляющие агрегатные функции по всей таблице, всегда возвращают одну строку, содержащую итоговые данные. Поэтому в списке выражений, следующим за словом SELECT, могут быть *только агрегатные функции* (или выражения на основе агрегатных функций).

Группировка и агрегатные функции

Агрегатные функции могут вычисляться как по всей таблице, так и по отдельным группам строк, в последнем случае над таблицей выполняется операция *группировки*.

При группировке формируются группы с одинаковыми значениями в столбце группировки (или нескольких столбцах), запрос возвращает столько строк, сколько получилось групп. Фактически количество групп совпадает с количеством уникальных значений в столбце группировки, поэтому группировку принято выполнять по столбцам, содержащим большое количество повторяющихся значений. Тогда количество групп будет значительно меньше, чем количество строк в таблице.

В базе данных, которую мы используем для демонстрационных примеров, наиболее подходящей для группировки является таблица оценок marks, в которой каждый столбец содержит большое количество повторяющихся значений и может быть использован в качестве столбца группировки.

Например, сгруппировав эту таблицу по столбцу cod_st, можно подсчитать различные итоговые данные по каждому студенту. Так, запрос:

```
SELECT cod_st, avg(mark) avg_mark, count(mark) count_mark
FROM marks
GROUP BY cod_st
```

возвращает средний балл и количество оценок для каждого студента. Вместе с итоговыми данными запрос возвращает коды студентов, для которых подсчитаны итоговые данные. Вывод столбца cod_st в приведенном запросе может быть выполнен корректно, поскольку при выполнении группировки в каждой группе оказались одинаковые коды студентов (но разные предметы и разные оценки).

Если выполнить группировку таблицы marks по столбцу cod_sub (код предмета), то можно подсчитать те же итоговые данные по каждому предмету. Но тогда в список вывода запроса, кроме агрегатных функций, можно включить только код предмета.

```
SELECT cod_sub, avg(mark) avg_mark, count(mark) count_mark
FROM marks
GROUP BY cod_sub
```

Обобщив примеры, выведем общее *правило для запросов с группировкой:*
в списке выражений, который следует за словом SELECT, могут быть *только выражения из фразы GROUP BY и агрегатные функции* (или выражения на их основе).

Условия отбора групп

После группировки можно отобрать группы, удовлетворяющие определенному условию. Для этих целей служит фраза HAVING..

Например, пусть требуется выбрать тех студентов, у которых средний бал >4. Текст запроса имеет вид:

```
SELECT cod_st, AVG(mark) avg_mark
FROM marks
GROUP BY cod_st
HAVING AVG(mark)>4
```

Следует отметить, что фраза HAVING может использоваться только после фразы GROUP BY.

4.4.2. Соединение таблиц в запросах

Запросы по одной таблице составляют малую долю всех запросов к базе данных. Например, запросы с группировкой, которыми завершалась предыдущая лекция, при использовании только одной таблицы оценок marks выдают числовые коды студентов или предметов, тогда как при формировании отчетов требуются фамилии студентов и названия предметов. Но эти данные хранятся в других таблицах (students и subjects).

Подобная ситуация является типичной для любой нормализованной базы данных, состоящей из большого числа таблиц. В связи с этим, одна из самых распространенных операций в запросах на выборку – операция соединения таблиц. Соединять таблицы в запросах можно различными способами, при этом могут получаться различные результаты, поэтому данная операция требует повышенного внимания.

Способы соединения таблиц в запросе SELECT

1. **Декартово произведение**

Это операция, при выполнении которой каждая строка одной таблицы соединяется (склеивается) с каждой строкой другой таблицы. Декартово произведение таблиц выполняется в том случае, если во фразе FROM присутствует список из нескольких таблиц, но не задается ни операция соединения JOIN, ни условие для соединения таблиц во фразе

WHERE. Обычно таблицы, которые соединяются посредством декартова произведения, не имеют общих столбцов.

Так, запрос вида:
SELECT … FROM таблица1, таблица2
возвратит количество строк, равное произведению количества строк первой таблицы на количество строк второй таблицы, поскольку склеит каждую строку таблицы1 с каждой строкой таблицы2. Если не выполнять операцию отбора столбцов, в результат войдут все столбцы таблицы1 и таблицы2, т.е. количество столбцов равно сумме количеств столбцов таблицы1 и таблицы2. В общем, размеры таблицы-результата оказываются весьма внушительными.

На практике декартово произведение таблиц используется крайне редко. Например, для нашей демонстрационной базы данных выполнять соединение всех студентов со всеми предметами (т.е. выполнять декартово произведение таблиц students и subjects) разумно только в том случае, если все учатся по единому плану и сдают экзамены по всем предметам, занесенным в таблицу subjects. Соединять всех студентов или все предметы со всеми оценками – вообще полная бессмыслица.

Иногда в базе данных встречаются таблицы, которые содержат всего одну строку для хранения каких-либо констант (название вуза, минимальный размер оплаты труда и т.д.). Такие таблицы можно подключать к любому запросу, используя операцию декартова произведения.

На практике декартовы произведения иногда возникают из-за ошибки в записи текста запроса. Основным способом соединения таблиц является операция внутреннего или естественного соединения.

2. **Внутреннее (естественное) соединение таблиц**

Таким способом можно соединять только таблицы, имеющие общие столбцы. При выполнении данной операции соединяются (склеиваются) только строки, имеющие общие значения в столбце связи. Как правило, таким способом соединяются таблицы, связанные отношением «один-ко-многим», а в качестве столбцов связи используются первичный ключ главной таблицы и внешний ключ подчиненной. Таким образом, те строки главной таблицы, для которых нет связанных строк в подчиненной таблице, при внутреннем соединении вообще не попадут в результат запроса.

В языке SQL имеются 2 способа реализации внутреннего соединения таблиц, оба этих способа являются равноценными и обычно приводят к одному и тому же плану исполнения запроса. Однако с точки зрения реляционной алгебры они используют различные операции, и тексты запросов несколько отличаются друг от друга:

а) **выборка из декартового произведения**

Начнем с примеров. Пусть требуется вывести фамилии всех студентов и их оценки. Текст запроса будет выглядеть так:

```
SELECT students.name_st, marks.mark
FROM students, marks
WHERE students.cod_st=marks.cod_st
```

б) **операция соединения [INNER] JOIN**

Тот же самый запрос, соединяющий студентов с их оценками, будет записан несколько по-другому:

```
SELECT students .name_st, marks.mark
FROM students JOIN marks
ON students.cod_st=marks.cod_st
```

Результаты запросов (а, б) будут абсолютно аналогичны и могут выглядеть примерно так:

name_st	mark
Иванов	5
Иванов	3
…	
Петров	4
Петров	5
Петров…	5

Каждая фамилия студента повторяется в результирующей таблице столько раз, сколько оценок получил данный студент. Если в таблице students есть, например, строка с фамилией Сидоров, который пока еще не получил ни одной оценки, в результирующей таблице этой фамилии вообще не будет. Так работает операция внутреннего соединения.

Обратим внимание на некоторые особенности приведенных выше примеров.

Во-первых, в тексте запросов используются составные имена столбцов, записанные с использованием точечной нотации:

имя_таблицы.имя_столбца

Использование составных имен позволяет избежать неоднозначности в записи имени столбца, поскольку разные таблицы могут содержать одноименные столбцы. В принципе, если имя какого-либо столбца уникально в пределах тех таблиц, которые указаны во фразе FROM, можно ограничиться и простым именем, но использование составных имен везде является более грамотным. Такие запросы и компилируются быстрее.

Во-вторых, в обоих приведенных выше запросах явно задано условие соединения в виде равенства столбцов связи (stu-

dents.cod_st=marks.cod_st). Казалось бы, можно сократить текст запроса, ведь в таблицах students и marks только один общий столбец cod_st. Однако соединение двух таблиц вовсе не обязательно должно выполняться только по первичному и внешнему ключу. Любые два столбца, совпадающие по типу, могут быть использованы в условии соединения таблиц. Разумеется, связывать строки, используя неключевые столбцы для связи, нужно с предельной осторожностью. Допустим, запрос, в котором устанавливается связь с помощью условия students.cod_st=marks.cod_sub, будет синтаксически правильным (он даже исполнится и возвратит результаты), но абсолютно бессмысленным. Будьте предельно внимательны при записи условий соединения!

Использование псевдонимов таблиц в запросах с соединением

Можно несколько сократить тексты приведенных выше запросов за счет использования коротких псевдонимов вместо довольно длинных имен таблиц.

Например, текст запроса, соединяющего студентов с оценками, может выглядеть так:

```
SELECT st.name_st, m.mark
FROM students st, marks m
WHERE st.cod_st=m.cod_st
```

или так:

```
SELECT st .name  st, m.mark
FROM students st JOIN marks m
ON st.cod_st=m.cod_st
```

В дальнейших примерах мы будем использовать псевдонимы таблиц.

Приведенные запросы не возвращают важной информации, по каким именно предметам студент Иванов или Петров получил те или иные оценки. Для того, чтобы в результатах запроса появилось название предмета, необходимо в тексте запроса добавить соединение еще с одной таблицей subjects:

a) вариант с выборкой из декартова произведения:

```
SELECT st.cod_st, st.name_st, s.name_sub, m.mark
FROM students st, marks m, subjects s
WHERE st.cod_st=m.cod_st AND s.cod_sub=m.cod_sub
```

b) вариант с использованием операции соединения:

```
SELECT st.cod_st, st.name_st, s.name_sub, m.mark
FROM students st JOIN marks m ON st.cod_st=m.cod_st
JOIN subjects s ON s.cod_sub=m.cod_sub
```

Результаты этих запросов опять одинаковы и предоставляют исчерпывающую информацию об успеваемости студентов в виде одной большой ненормализованной таблицы:

cod_st	name_st	name_sub	mark
1	Иванов	Математика	5
1	Иванов	Физика	4
…	…	…	
2	Петров	Физика	4
2	Петров	Информатика	5
2	Петров	История	4
…	…	…	…
3	Иванов	Математика	4
3	Иванов	Информатика	3
…	…	…	…

Как можно понять из текста запросов, приведенных выше, для каждой добавляемой в запрос новой таблицы необходимо добавить условие ее соединения с другой таблицей, в противном случае будет выполнена операция декартова произведения. В общем случае, если в запросе используется n таблиц, нужно записать n-1 условий их соединения.

Следующий пример запроса выводит ФИО, код студента и средний балл (обратим внимание на то, что группировку придется выполнять сразу по двум столбцам, чтобы выполнилось правило для запросов с группировкой, которое было рассмотрено в предыдущей лекции).

```
SELECT st.cod_st, st.name_st, AVG(m.mark) avg_mark
FROM students st, marks m WHERE st.cod_st=m.cod_st
GROUP BY st.cod_st, st.name_st
```

Есть и еще один вариант обхода правила для запросов с группировкой:

```
SELECT st.cod_st, MAX(st.name_st) name_st, AVG(m.mark) avg_mark
FROM students st, marks m WHERE st.cod_st=m.cod_st
GROUP BY st.cod_st
```

В приведенном примере использование агрегатной функции MAX(st.name_st) выглядит искусственным, с таким же успехом можно использовать и функцию MIN, однако любая из этих функций позволит выполнить группировку только по одному столбцу cod_st.

3. **Внешнее соединение таблиц (операция OUTER JOIN)**

При внешнем соединении, в отличие от внутреннего, в результат выборки попадают не только все связанные строки обеих таблиц, но и строки одной из таблиц (или обеих), для которых нет связанных в другой

таблице. Недостающим значениям столбцов другой таблицы при этом присваивается значение NULL.

Возможны три варианта внешнего соединения двух таблиц (выборка дополняется строками таблицы, стоящей слева от слова JOIN, или таблицы, стоящей справа, или обеих сразу), поэтому различают три вида внешних соединений:

LEFT [OUTER] JOIN – левое внешнее соединение
RIGHT [OUTER] JOIN – правое внешнее соединение
FULL [OUTER] JOIN – полное внешнее соединение.

Например, перепишем предыдущий запрос (код, фамилия и средний балл студента), используя внешнее соединение таким образом, чтобы студенты, у которых вообще нет оценок, попали бы в список вывода с NULL-значениями среднего балла.

```
SELECT st.cod_st, st.name_st, AVG (m.mark) avg_mark
FROM students st LEFT JOIN marks m ON st.cod_st=m.cod_st
GROUP BY st.cod_st, st.name_st
```

Мы уже знаем, что внутреннее соединение таблиц часто оформляется в тексте запроса, как выборка из декартова произведения. Можно ли использовать этот способ для внешнего соединения?

В некоторых СУБД можно, но, как и все нестандартные конструкции, это плохо влияет на переносимость запроса. Все же приведем вариант записи внешнего соединения таблиц, используя синтаксис СУБД Oracle (на примере того же самого запроса - код, фамилия и средний балл студента).

```
SELECT st.cod_st, st.name_st, AVG (m/mark) avg_mark
FROM students st, marks m
WHERE st.cod_st=m.cod_st (+)
GROUP BY st.cod_st, st.name_st
```

Используемая здесь синтаксическая конструкция (+) добавляет фиктивные строки в таблицу marks для тех студентов, у которых нет оценок, при этом в столбец mark помещается значение NULL.

Рассмотрим некоторые особенности использования функции COUNT в запросах с внешим соединением таблиц. Пусть требуется вывести количество оценок для каждого студента из таблицы students. Если студент еще не имеет ни одной оценки, должно быть выведено количество 0. Текст запроса, использующий операцию LEFT JOIN:

```
SELECT st.cod_st, st.name_st, COUNT(m.mark) count_mark
FROM students st LEFT JOIN marks m ON st.cod_st=m.cod_st
GROUP BY st.cod_st, st.name_st
```

Использование конструкции COUNT(m.mark) позволит получить правильные результаты и вывести значение 0 для студентов, не имеющих оценок. Однако, если использовать в этом же запросе COUNT(*), то для

студентов, не имеющих оценок, будет выведено количество 1 (!) и их нельзя будет отличить от студентов, которые имеют одну оценку.

Такие результаты вполне согласуются с правилами стандарта SQL, поскольку операция внешнего соединения добавляет в таблицу marks фиктивную строку, а функция COUNT(*) ее добросовестно подсчитывает. Функция COUNT(m.mark) учитывает, что в фиктивных строках, а также в реальных строках, где преподаватель не выставил оценку, значение столбца m.mark равно NULL.

Совсем интересные результаты получатся при использовании конструкции COUNT(DICTINCT m.mark). В этом случае будет подсчитано количество различных оценок каждого студента (например, для круглых отличников это значение равно 1 – одни пятерки)

4. Самосоединения

Это соединение таблицы со своей копией. Подобные соединения используются сравнительно редко, но при решении некоторых задач могут оказаться полезными.

Например, пусть требуется вывести всех однофамильцев, т.е. фамилии, которые встречаются в таблице студентов более одного раза:

```
SELECT DISTINCT s1.name_st FROM students s1, students s2
WHERE s1.name_st=s2.name_st AND s1.cod_st<>s2.cod_st
```

Ключевое слово DISTINCT здесь добавлено на тот случай, если в таблице встречаются фамилии, повторяющиеся более двух раз (такие фамилии несколько раз попадут в результат запроса).

Следует отметить, что ту же задачу можно решить более эффективно, используя запрос:

```
SELECT name_st FROM students
GROUP BY name_st
HAVING COUNT(name_st) > 1
```

Так что будем считать первый из приведенных вариантов лишь иллюстрацией к операции самосоединения таблиц в запросах на выборку. В связи с этим стоит отметить, что часто вопрос об эффективности того или иного запроса для конкретной СУБД приходится решать экспериментально, поэтому при решении задачи желательно рассмотреть несколько вариантов текста SQL-запроса с целью выбора наиболее эффективного.

4.4.3. Вложенные запросы

Это запросы, которые содержатся в теле другого запроса. Они могут использоваться во фразах FROM, WHERE и HAVING, а также в списке после слова SELECT, создавая, таким образом, вычисляемый столбец.

Вложенные запросы – прекрасное средство разработки сложных запросов выборки данных, которые позволяют применить алгоритмический

подход к решению задачи, представив сложную задачу выборки в виде последовательности более простых задач, каждую из которых можно оформить в виде вложенного запроса.

Перечислим несколько основных правил, которые должны соблюдаться при использовании в запросе вложенных запросов:

- тело вложенного запроса всегда заключается в скобки
- вложенные запросы могут содержать другие вложенные запросы, при этом выполнение запроса всегда начинается с самого «глубокого» вложенного запроса и заканчивается внешним запросом
- во вложенном запросе не следует использовать фразу ORDER BY, поскольку сортировка результатов должна быть выполнена один раз после выполнения всего запроса целиком

Другие особенности использования вложенных запросов рассмотрим немного позже. Перейдем к примерам.

Вложенные запросы во фразе FROM

Это одна из самых простых и понятных ситуаций использования вложенных запросов. Поскольку любой запрос возвращает результаты в виде таблицы, то использование конструкции

SELECT …. FROM (SELECT …)

полностью равнозначно

SELECT …. FROM имя_таблицы

Например, пусть требуется найти значения минимального и максимального среднего балла студентов. Для решения этой задачи сначала необходимо подсчитать средние баллы по каждому студенту, что можно с успехом выполнить во вложенном запросе. Результаты вложенного запроса затем используются во внешнем запросе как обычная таблица из одного столбца avg_mark (задание псевдонима во вложенном запросе обязательно), над которой производится вычисление агрегатных функций MIN и MAX.

```
SELECT MIN (avg_mark) min_avg_mark, MAX(avg_mark) max_avg_mark
FROM (SELECT AVG (mark) avg_mark FROM marks GROUP BY cod_st)
```

Если во вложенном запросе выполнить группировку таблицы оценок по предмету (cod_sub), будут получены минимальный и максимальный средний балл по предметам.

Вложенные запросы как альтернатива соединению таблиц в запросах

Этот вариант использования вложенного запроса во многих (но не во всех!) случаях позволяет избежать операции соединения таблиц. Текст такого запроса выглядит понятно и логично, эффективность для многих СУБД сравнима с использованием операции соединения, но существенного ускорения запроса таким способом не добиться. Тем не менее, как альтернативный вариант соединению, он заслуживаеи внимания.

Например, пусть требуется выяснить оценку студента Иванова по математике (если таких студентов несколько – все оценки). Можно легко решить эту задачу, соединив таблицы students, subjects и marks и выполнив выборку из полученной в результате соединения таблицы. Однако, логичным представляется и такой вариант: найти коды всех Ивановых из таблицы students, найти код математики из таблицы subjects, на последнем этапе найти оценку (или несколько оценок) в таблице marks. Получаем такой текст запроса с двумя вложенными запросами:

```
SELECT mark FROM marks
WHERE cod_st IN
(SELECT cod_st FROM students WHERE name_st='Иванов')
AND cod_sub IN
(SELECT cod_sub FROM subjects WHERE name_sub='Математика')
```

Обратим внимание на использование операции IN (принадлежность значения множеству). Вложенные запросы, которые используются для извлечения кода студента и кода предмета, в общем случае, возвращают множество значений. Хотя в таблице предметов название предмета является уникальным столбцом, второй вложенный запрос может возвратить пустое множество при отсутствии предмета с названием 'Математика'. Поэтому применение операции IN в данной ситуации является правильным.

Еще один пример. Пусть требуется выяснить средний балл всех студентов по фамилии Иванов. Логика рассуждений та же, что и в предыдущем случае – сначала найти коды Ивановых, а затем выполнить всю оставшуюся работу, используя только таблицу оценок:

```
SELECT cod_st, AVG (mark) avg_mark FROM marks
WHERE cod_st IN
(SELECT cod_st FROM students WHERE name_st='Иванов')
GROUP BY cod_st
```

Вложенные запросы в условиях отбора (WHERE и HAVING)

Эти запросы по своей логике ничем не отличаются от запросов, приведенных выше, поэтому просто приведем примеры.

Пусть требуется вывести фамилии самых молодых студентов (т.е. тех, у которых самая поздняя дата рождения). Этот запрос состоит из двух этапов: сначала нужно найти самую позднюю дату рождения (эту задачу решит вложенный запрос), а затем всех студентов, которые имеют найденную дату рождения.

```
SELECT name_st FROM students WHERE born=
(SELECT MAX(born) FROM students)
```

Здесь знак равенства в условии WHERE уместен, поскольку вложенный запрос гарантированно возвратит одно непустое значение, если в таблице students имеется хотя бы одна строка (на пустой таблице этот

запрос также корректно работает, возвращая сообщение «строки не выбраны»).

Следующий пример несколько объемнее, но использует ту же логику разбиения задачи на несколько этапов.

Пусть требуется найти студентов, которые имеют максимальный (или минимальный) средний балл. Запрос, который находит максимальный средний балл студента, приводился выше, осталось только продолжить логику рассуждений. Зная максимальный средний балл, по таблице оценок можно найти коды студентов, имеющих такой средний балл (здесь потребуется использование фразы HAVING), а по кодам студентов в самом внешнем запросе можно отыскать в таблице студентов их фамилии.

В результате получили «многоэтажный», но довольно простой по логике, запрос:

```
SELECT name_st FROM students WHERE cod_st IN
(SELECT cod_st FROM marks GROUP BY cod_st HAVING AVG(mark)=
 (SELECT MAX(avg_mark) max_avg_mark FROM
   (SELECT AVG (mark) avg_mark FROM marks GROUP BY cod_st
   )
  )
)
```

Вложенные запросы в качестве вычисляемых столбцов

Такое использование вложенных запросов приводит к очень интересной конструкции

SELECT …(SELECT …) FROM …

Она поддерживается большинством СУБД.

В качестве примера приведем задачу вычисления средних баллов всех студентов, для которой уже было представлено несколько вариантов запросов. Приведем еще один, не самый эффективный, но правильный.

```
SELECT cod_st, name_st,
(SELECT AVG(mark) FROM marks WHERE cod_st=students.cod_st)
avg_mark
FROM students
```

В данном примере avg_mark является вычисляемым столбцом, для которого в качестве выражения используется вложенный запрос, гарантированно возвращающий одно значение для каждой строки внешнего запроса.

В последнем приведенном примере можно обратить внимание на одно важное обстоятельство. Вложенный запрос, использующий только одну таблицу marks, тем не менее, содержит во фразе WHERE столбец из другой таблицы students.mark.

Дело в том, что таблица students используется во внешнем запросе, поэтому ее данные разрешено использовать и во всех вложенных запросах. Обсудим эту проблему подробнее.

Кореллированные и некореллированные вложенные запросы

Вложенные запросы, которые не используют данных внешнего запроса, называются некореллированными. Все приведенные ранее примеры, кроме самого последнего, содержали некореллированные вложенные запросы. Такие вложенные запросы выполняются один раз и их результаты используются во внешнем запросе в качестве констант.

Некореллированные вложенные запросы в большинстве случаев не приводят к снижению производительности, если их использовать аккуратно.

Совсем иначе дело обстоит с запросами, которые используют данные внешнего запроса. Такие запросы называются кореллированными. Особенность кореллированных запросов состоит в том, что они выполняются для каждой строки внешнего запроса. Это приводит к существенным затратам времени, поэтому в документации к различным СУБД приводятся рекомендации использовать кореллированные запросы только в случае крайней необходимости.

Тем не менее, раз такая возможность в языке SQL имеется, пользоваться ею можно, но с предельной осторожностью.

В следующих разделах будет приведено несколько примеров кореллированных запросов и показано, как можно решить те же задачи более эффективными способами.

Использование ключевых слов ALL и ANY с вложенными запросами

Конструкции ALL(SELECT ...) и ANY(SELECT...) применяются к вложенным запросам, возвращающим множество строк, употребляются, как правило, во фразах WHERE и HAVING, и означают буквально следующее:

ALL (SELECT...) – все строки вложенного SELECT

ANY (SELECT...) – хотя бы одна из строк вложенного SELECT

В некоторых случаях употребление этих слов позволяет сформулировать запрос просто и точно.

Проиллюстрируем это на примере. Пусть требуется вывести студентов, у которых все оценки – только четверки (никаких других оценок нет). Использование ключевого слова ALL позволяет решить эту задачу «в лоб»:

```
SELECT cod_st,name_st FROM students
WHERE 4=ALL(SELECT mark FROM marks WHERE
cod_st=students.cod_st)
```

Если же требуется найти студентов, у которых имеется хотя бы одна четверка, можно прибегнуть к помощи ключевого слова ANY:

```
SELECT cod_st,name_st FROM students
WHERE 4=ANY(SELECT mark FROM marks WHERE cod_st=students.cod_st)
```

К сожалению, в данных примерах получились кореллированные вложенные запросы, так что прозрачность формулировок запроса идет в ущерб производительности.

Можно привести примеры более эффективных запросов, решающих те же самые задачи.

Например, чтобы найти студентов, у которых все оценки четверки, достаточно сообразить, что у таких студентов и минимальная, и максимальная оценки равны 4:

```
SELECT st.cod_st, st.name_st FROM students st, marks m
WHERE st.cod_st= m.cod_st
GROUP BY m.cod_st
HAVING MIN(m.marks)=4 AND MAX(m.marks)=4
```

А найти студентов, у которых есть хотя бы одна четверка, можно, например, таким запросом:

```
SELECT DISTINCT st.cod_st, st.name_st FROM students st, marks m
WHERE st.cod_st= m.cod_st AND m.mark=4
```

Функция EXISTS

Эта функция служит для проверки результатов вложенного запроса на пустоту.

EXISTS(SELECT…) возвращает значение «истина», если вложенный SELECT возвращает хотя бы одну строку, и «ложь» в противном случае.

Функция NOT EXISTS (SELECT …) противоположна функции EXISTS.

Например, пусть требуется вывести всех студентов, у которых нет оценок.

```
SELECT cod_st, name_st FROM students
WHERE NOT EXISTS
(SELECT cod_st FROM marks WHERE cod_st=students.cod_st )
```

Приведенный текст запроса прост для понимания, но есть способы решить ту же задачу более эффективно, используя некореллированный вложенный запрос:

```
SELECT cod_st, name_st FROM students
WHERE cod_st NOT IN
(SELECT DISTINCT cod_st FROM marks)
```

или внешнее соединение таблиц:

```
SELECT st.cod_st, st.name_st FROM students st
LEFT JOIN marks m ON st.cod_st=m.cod_st
WHERE m.mark IS NULL
```

4.4.4. Комбинированные запросы

Над результатами запросов можно выполнять обычные операции над множествами.

Эти операции выполняются только над запросами, которые возвращают одинаковое количество столбцов одинакового типа.

Стандартом установлены следующие обозначения для операций:
UNION [ALL]-объединение запросов
INTERSECT [ALL] –пересечение запросов
ECXEPT (MINUS в Oracle) [ALL] – разность запросов

Ключевое слово ALL в данном контексте запрещает удаление дубликатов из результатов операций (тем самым повышается эффективность, поскольку удаление дубликатов – достаточно медленная операция).

```
SELECT name_st FROM students1
UNION ALL //всех однофамильцев, не удаляет дубликаты
SELECT name_st FROM students2
//если нет all удалит всех однофамильцев!!

SELECT name_st FROM students1
INTERSECT //выводит только однофамильцев
SELECT name_st FROM students2

SELECT name_st FROM students1
MINUS //все students1 если ни один не входит в students2
SELECT name_st FROM students2
```

4.5. Представления (VIEW)

4.5.1. Понятие представления

Представления (другие варианты перевода – просмотры, виды) - это именованные запросы на выборку, сохранённые в БД, которые при любом обращении к ним по имени создают виртуальную таблицу, наполняя ее актуальными данными из БД.

Для того чтобы лучше понять, для чего нужны представления и как они работают, сначала рассмотрим, как СУБД обрабатывает обычный SQL-запрос на выборку. Схематическое изображение этого процесса приведено на рис. 4.2.

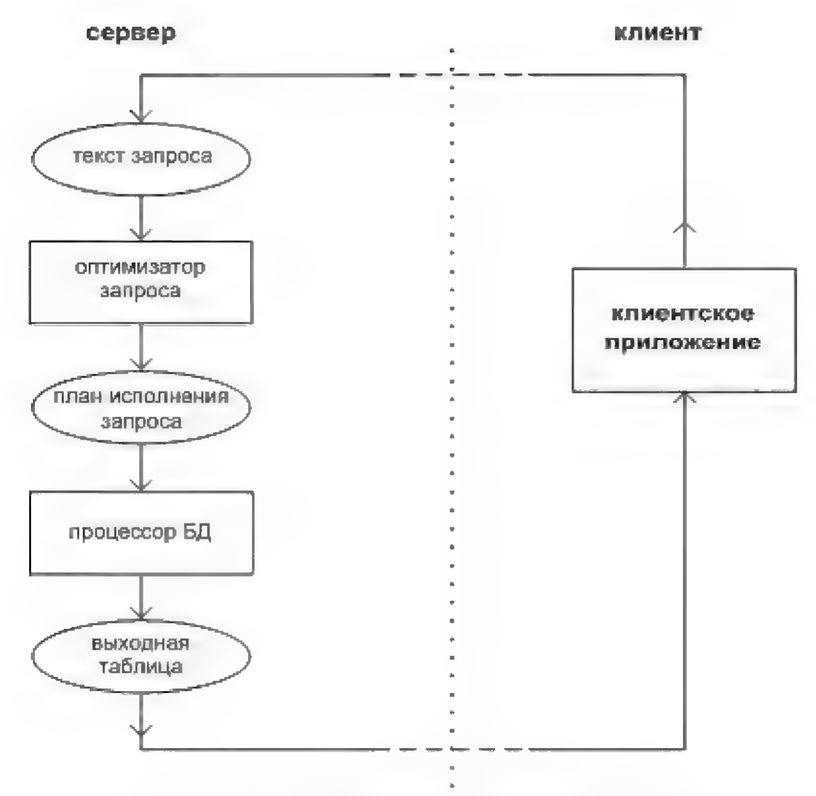


Рис.4.2. Порядок обработки SQL-запроса SELECT ... .

Исходный текст запроса, переданный по сети из клиентского приложения, сначала подвергается проверке на правильность всех синтаксических конструкций и наличие всех таблиц и столбцов с именами, заданными в тексте запроса. Для запроса, который признан правильным, затем формируется *план его исполнения,* представляющий собой описание (во внутреннем формате СУБД) наиболее оптимального способа реализации тех реляционных операций, которые содержатся в тексте запроса. Все эти действия выполняет специальный компонент СУБД, который называется *оптимизатором запроса* (Query Optimizer), а сам этап формирования плана исполнения запроса называют компиляцией по аналогии с первым этапом обработки программы, написанной на любом языке програмирования. Правда, план исполнения запроса не является объектным кодом, который формирует компилятор с языка Pascal или C.

Оптимизатор запроса передает план исполнения запроса другому компоненту СУБД, который называется *процессором базы данных* (или

процессором SQL). Процессор БД исполняет все необходимые действия по извлечению и обработке данных. В результате формируется таблица с выходными данными, которая возвращается клиенту в ответ на его запрос.

Запросы на выборку, которые необходимо выполнять регулярно, нет смысла пересылать по сети и компилировать каждый раз, как только клиенту потребуется соответствующая выборка данных. Разумно постоянно хранить в базе данных тексты таких запросов вместе с планами их исполнения.

Может возникнуть вопрос, зачем хранить исходные тексты запросов, а не ограничиться только планами их исполнения? Дело в том, что при наличии исходного текста имеется возможность перестройки плана исполнения запроса, если старый план окажется уже не самым оптимальным (во всяком случае, СУБД Oracle такую возможность поддерживает).

К сожалению, подробный анализ работы оптимизатора запросов не входит в рамки настоящего курса. Самое главное, что требуется уяснить, - то, что представление не содержит никаких данных, в отличие от таблицы, но с точки зрения клиентского приложения представление почти ничем не отличается от таблицы. Точнее, представление является виртуальной таблицей, с которой в большинстве практических применений можно работать так же, как с реально существующей на диске таблицей.

Сказанное означает, что во всех запросах на выборку SELECT можно использовать имя представления везде, где можно использовать имя таблицы (т.е. при формулировании запросов на выборку пользователь может даже не знать, чем он пользуется – таблицей или представлением). Более того, некоторые представления (но не все) можно использовать даже в запросах INSERT, DELETE и UPDATE, при этом будут внесены соответствующие изменения в реальные таблицы.

Использование представлений имеет глубокий смысл, который формулируется в правиле №7 Кодда для реляционных баз данных:

«База данных должна быть доступна конечным пользователям только через представления».

Иными словами, механизм представлений позволяет предоставлять каждому пользователю только ту часть базы данных, которая ему действительно требуется для работы (внешнюю схему), скрывая от многочисленных пользователей концептуальную схему, доступную только администратору базы данных.

Для разработчика использование представлений упрощает разработку сложных SQL-запросов, которые можно строить на основе представлений и отлаживать по частям.

Однако не следует злоупотреблять замечательными возможностями, которые предоставляют представления. Не следует забывать, что для ма-

териализации представлений всегда выполняется SQL-запрос, на выполнение которого требуется время.

4.5.2. Создание и удаление представлений

Поскольку представление является объектом базы данных, для его создания используется стандартная команда языка DDL CREATE…

```
CREATE VIEW имя [(список_столбцов)]
AS
SELECT …// любой запрос на выборку
```

Команда проста, поскольку все, что нужно для создания представления – его имя и запрос на выборку, который лежит в основе данного представления. Список столбцов представления должен быть указан явно, если в запросе присутствуют вычисляемые столбцы без указания псевдонима. В остальных случаях столбцы представления получат такие же имена, как используемые в запросе столбцы таблиц. Все же рекомендуется задавать список столбцов в явном виде.

Например, создадим представление, содержащее столбцы «код студента, фамилия студента, его средний балл» :

```
CREATE VIEW stud_mark (cod_st, name_st, avg_mark)
AS
SELECT  st.cod_st, st.name_st, AVG(m.mark)
FROM students st LEFT JOIN marks m
ON st.cod_st=m.cod_st
GROUP BY st.cod_st, st.name_st
```

При создании представления пользователю не возвращается виртуальная таблица, которую он обычно получает при выполнении SELECT… Вместо этого он получит лаконичное сообщение типа «Представление создано».

Для того, чтобы материализовать представление, т.е. получить данные в виде виртуальной таблицы, необходимо выполнить запрос SELECT к представлению так же, как к обычной таблице:

```
SELECT * FROM stud_mark
```

Можно написать и любой другой запрос на выборку к представлению:

```
SELECT cod_sub, avg_mark FROM stud_mark
WHERE cod_st=123
```

Удаляется представление также стандартными средствами DDL:

```
DROP VIEW имя_представления
```

Выполнив последовательно команды DROP VIEW…, а затем снова CREATE VIEW … с тем же самым запросом SELECT, мы не напрасно потеряем время. Во многих случаях СУБД сформирует новый план ис-

полнения того же самого запроса в соответствии с изменившимся наполнением таблиц или появлением новых индексов. Это приведет к сокращению времени материализации представления.

СУБД Oracle позволяет изменить план исполнения запроса без удаления представления, используя команду:

```
ALTER VIEW имя RECOMPLILE
```

4.5.3. Обновление представлений

На некоторые представления можно писать запросы UPDATE, DELETE и INSERT, как на обычные таблицы. При выполнении таких запросов реально все изменения вносятся в физические таблицы.

Такие представления называются обновляемыми. Согласно стандарту SQL, обновляемыми являются представления, основанные на запросах:

1) только по одной таблице
2) запрос не должен содержать ключевых слов DISTINCT, GROUP BY, HAVING
3) не должен содержать вложенных запросов
4) не должен быть комбинированным, т.е. не должен содержать операций UNION, INTERSECT, EXCEPT (MINUS).

Oracle, дополнительно, не позволяет обновлять запросы с сортировкой результатов.

Таким образом, обновляемых представлений не так и много:

- представления с отбором столбцов (вертикальные представления)
- представления с отбором строк (горизонтальные представления)
- представления с отбором строк и столбцов (смешанные представления).

Например, создадим горизонтальное представление на основе выборки из таблицы оценок marks, содержащее только оценки по Математике.

Предварительно мы выяснили из таблицы предметов subjects, что математика имеет код 1.

Тогда для создания представления потребуется выполнить команду:

```
CREATE VIEW mark_1
AS
SELECT  * FROM  marks
WHERE cod_sub=1
```

Это представление будет обновляемым, поэтому преподаватель математики, которому разрешено обновление этого представления, может изменить любую (одну) оценку, например, таким запросом:

```
UPDATE mark_1 SET mark=5 WHERE cod_st=123
```

Однако запрос:

```
UPDATE mark_1 SET cod_sub=2 WHERE cod_st=123
```

который также, как и первый, является разрешенным, приведет к неожиданным последствиям. Обновленная строка окажется за пределами представления mark_1 и запрос:

```
SELECT * FROM mark_1
```

эту строку не покажет.

Если такой побочный эффект нежелателен, в команду CREATE VIEW … следует добавить дополнительную фразу WITH CHECK OPTION:

```
DROP VIEW mark_1
CREATE VIEW mark_1
AS
SELECT  * FROM  marks WHERE cod_sub=1
WITH CHECK OPTION
```

Теперь при любой попытке изменить код предмета или добавить предмет с кодом, отличным от единицы, будет выдано сообщение об ошибке.

Если говорить о данном конкретном примере, то можно было бы поступить еще проще: не включать столбец cod_sub в представление, тогда у преподавателя математики просто не будет никакой возможности изменить (нечаянно или преднамеренно) код своего предмета.

4.5.4. Стандартные представления словаря данных Oracle

Напомним, что словарь данных СУБД – это совокупность служебных таблиц, в которых хранится исчерпывающая информация обо всех объектах базы данных (метаданные). Доступ к словарю данных Oracle осуществляется только через представления, названия которых формируются по определенным правилам:

dba_название (служебная информация для АБД)
user_ название (информация об объектах конкретного пользователя)
all_ название (полная информация)

Примеры представлений:

user_tables (информация о таблицах)
user_views (информация о представлениях)
user_triggers (информация о триггерах)
dba_users (имена пользователей)
all_objects (информация обо всех объектах)

Например, с помощью команды

```
SELECT table_name FROM user_tables
```

можно получить имена всех таблиц, созданных конкретным пользователем.

Особо хочется отметить представление ***user_errors***, которое содержит сведения об ошибках, полученных при компиляции хранимого кода. Оно будет использоваться в следующей лекции.

4.6. Хранимый код. Триггеры

4.6.1. Процедурные расширения языка SQL

Как мы показали в предыдущих лекциях, язык SQL является очень мощным языком манипулирования данными, однако для решения сложных задач обработки данных ему не хватает управляющих конструкций, имеющихся в универсальных языках программирования. В связи с этим многие СУБД имеют процедурные расширения этого языка, которые представляют собой полноценный язык программирования, поддерживающий возможность использования в нем операторов SQL. На таком языке можно писать процедуры и функции, постоянно хранящиеся в базе данных и исполняемые в среде СУБД.

К сожалению, на настоящий момент ситуация такова, что каждая СУБД поддерживает свой собственный язык процедурного расширения SQL, что усложняет задачи переносимости программного обеспечения. Поэтому те примеры, которые будут приведены в этой лекции, используют процедурный язык PL/SQL, поддерживаемый Oracle, и работают только в среде этой СУБД. Из других процедурных расширений наиболее близок к PL/SQL язык СУБД PostgreSQL, который называется PLG/SQL. Используемое в Microsoft SQL Server процедурное расширение Transact-SQL по синтаксическим конструкциям отличается от PL/SQL, но по семантике является близким. Во всяком случае понимание логики разработки хранимого кода на PL/SQL поможет легко освоить и любое другое процедурное расширение.

Для дальнейшего изложения необходимо привести минимальные сведения по конструкциям языка PL/SQL. По синтаксису он наиболее близок языку программирования ADA, конструкции его очень логичны. Основной программной единицей является блок – совокупность операторов, заключенная в операторные скобки BEGIN ... END. При выполнении процедурных действий в блоке, как правило, необходимы переменные для хранения промежуточных значений. Объявление переменных предшествует блоку и образует секцию объявления, которая начинается ключевым словом DECLARE. Для переменных поддерживаются те же типы, что и для столбцов таблиц Oracle. Имеется возможность указать тип переменной, явно ссылаясь на определенный столбец или таблицу.

Например:

```
DECLARE
 x INTEGER;
 fio students.name_st%TYPE;
 s subjects%TYPE;
BEGIN
 . . .
END
```

В приведенном примере для переменной x тип указан явно, переменная fio имеет такой же тип, как столбец name_st таблицы students, переменная s имеет тип запись (RECORD), структура которой идентична строке таблицы subjects (содержит два поля cod_sub и name_sub).

Операторы языка PL/SQL

1. Оператор присваивания

```
переменная:= выражение;
```

2. Условный оператор

```
IF условие THEN
  оператор1; оператор2;
  ...
[ELSIF условие THEN
  оператор3; оператор4;
  ...]
[ELSE
  оператор5; оператор6;
  ...]
END IF;
```

3. Операторы цикла

Бесконечный цикл, условие выхода задается в теле цикла

```
LOOP
 оператор1; оператор2;
 ...
 EXIT WHEN условие выхода из цикла;
END LOOP;
```

Цикл с предусловием

```
WHILE условие LOOP
 оператор1; оператор2;
 ...
END LOOP;
```

Цикл с параметром

```
FOR параметр IN (множество значений) LOOP
  оператор1; оператор2; ...
END LOOP;
```

Множество значений параметра обычно задается в виде диапазона (начальное_значение .. конечное_значение)

5. Оператор безусловного перехода:

```
GOTO метка;
...
метка: оператор;
```

6. Оператор возврата из процедуры/функции

RETURN;

RETURN выражение; (только функции)

7. Комментарии

Однострочный комментарий обозначается так:

-- далее следует текст комментария до конца строки

Многострочный комментарий обозначается аналогично языку С:

/* текст комментария может располагаться где угодно и занимать сколько угодно строк */

8. Средства для обработки исключительных ситуаций

Общий подход к обработке исключительных ситуаций состоит в том, что для каждой ситуации определяется ее обработчик и при возникновении ситуации выполняется код обработчика.

В PL/SQL предусмотрена возможность как обрабатывать стандартные ситуации, так и вводить собственные исключительные ситуации. Предусматриваются также средства генерации исключительных ситуаций.

Для стандартных исключительных ситуаций существует большое число предустановленных имен. Пользовательская исключительная ситуация вводится объявлением переменной типа EXCEPTION. Генерация исключений из программы выполняется в PL/SQL оператором RAISE.

Обработчик исключений в PL/SQL, общий для всех исключений, составляет отдельную часть блока PL/SQL, начинающуюся со слова EXCEPTION и содержащую набор операторов WHEN, распознающих типы исключений и задающих действия, выполняемые по каждому типу (возможны любые действия, которые можно запрограммировать средствами PL/SQL).

Например:

```
DECLARE
 err EXCEPTION;
BEGIN
INSERT INTO … VALUES … ;
SELECT ... INTO …;
 IF ... THEN
  RAISE err;
 ...
EXCEPTION

 WHEN DUPLICATE_KEYS THEN
   ...
 WHEN TOO_MANY_ROWS THEN
   ...
 WHEN err THEN
   ...
 WHEN OTHERS THEN
   ...
END;
```

В приведенном примере имеется два предопределенных имени для стандартных исключительных ситуаций, возникающих в процессе выполнения команд INSERT и SELECT …INTO и предопределенное имя OTHERS, предназначенное для обозначения любой другой исключительной ситуации, которую процедура отдельно не обрабатывает. Переменная err предназначена для возбуждения пользовательской исключительной ситуации при помощи оператора RAISE.

Как видим из приведенного примера, блок PL/SQL может содержать команды языка SQL, которые органично сочетаются с операторами языка высокого уровня. Эта тема заслуживает отдельного обсуждения, поскольку механизм встраивания команд в язык высокого регламентируется стандартом SQL и практически одинаков во всех СУБД.

4.6.2. Использование команд SQL в хранимом коде

Команды INSERT, DELETE и UPDATE используются в программе на PL/SQL в качестве отдельных операторов наряду с другими операторами языка. В данных командах разрешено использовать переменные программы везде, где по правилам SQL используются константы, что делает данные команды более гибкими, чем при их использовании в интерактивном режиме. Для обработки исключительных ситуаций, которые могут возникнуть в случае, когда какая-либо из этих команд нарушает

целостность данных, существует большое количество стандартных предопределенных имен. Например, приведенная в предыдущем примере ситуация DUPLICATE_KEYS возникает при нарушении ограничения уникальности (и в первичном ключе в том числе).

Проблема возникает при встраивании в процедурный язык оператора SELECT. Результатом оператора SELECT является множество строк, а процедурный язык ориентирован в основном на обработку последовательностей. Для преодоления этого противоречия в стандарт SQL введен механизм курсора, который реализован и в PL/SQL Oracle. Курсор представляет собой результат выборки из базы данных, который предназначен для дальнейшей построчной обработки.

Различают неявный и явный курсоры. Неявный курсор можно использовать только в том случае, если запрос на выборку возврашает ровно одну строку. Тогда этот результат можно поместить в обычные переменные, используя расширенный синтаксис команды SELECT:

SELECT список_выражений INTO список_переменных ...
остальная часть оператора SELECT

Количество переменных в списке и их типы должны в точности соответствовать списку выражений оператора SELECT.

SELECT mark INTO m FROM marks WHERE cod_st=c_st AND cod_sub=c_s

Значения переменных c_st и c_s задаются заранее. Если существуют студент и предмет с такими значениями кодов, запрос вернет ровно одно значение оценки и разместит его в переменной с именем m.

При выполнении команды SELECT ... INTO … в различных случаях ее применения могут возникнуть две разные исключительные ситуации:

- TOO_MANY_ROWS возникает в том случае, если запрос SELECT вместо одной строки возвращает несколько строк (в этом случае возвращаемые данные невозможно разместить в заданном списке переменных)
- NO_DATA_FOUND возникает в том случае, если запрос SELECT вообще не возвращает данных. Тогда переменные в списке не могут получить никаких значений.

При наличии обработчиков для каждой из указанных ситуаций применение неявного курсора является простым и вполне безопасным способом обработки результатов однострочной выборки из базы данных. Примеры практического использования данной конструкции мы приведем в следующем разделе.

Явный курсор является более универсальным средством обработки произвольной выборки из базы данных. Он должен быть явно объявлен в разделе DECLARE. В объявлении курсора определяется его имя и запрос, на котором он основан.

DECLARE CURSOR имя_курсора IS SELECT ...далее идет запрос на выборку

Например:

```
DECLARE CURSOR cur IS
  SELECT name_st FROM students WHERE name_st LIKE 'A%'
```

Следует отметить, что приведенное выше объявление курсора, принятое в Oracle, не совсем соответствует стандарту. Согласно стандарту объявление курсора выглядит так:

имя_курсора CURSOR FOR SELECT ….

Все остальные операции с курсором соответствуют стандарту.

Объявление курсора не является выполнимым оператором. Выборка, заданная в объявлении курсора, выполняется только при его открытии.

Например:

```
  OPEN CURSOR cur
```

После открытия курсора можно последовательно выбирать строки курсора, используя оператор FETCH. Например:

```
  FETCH cur INTO fio
```

Переменная fio должна быть предварительно объявлена, например, так:
fio students.name_st%TYPE;

Каждое следующее выполнение FETCH выбирает значение столбцов из следующей строки выборки в переменные заданного списка. Оператор FETCH, как правило, применяется в цикле. Например:

```
LOOP
  FETCH cur INTO fio;
  ...
  EXIT WHEN NOT cur%FOUND;
END LOOP;
```

или

```
FETCH cur INTO fio;
WHILE  cur%FOUND LOOP
FETCH cur INTO fio;
…
END LOOP;
```

После того, как выбраны все нужные строки, курсор должен быть закрыт. Например:

```
  CLOSE cur
```

Цикл по курсору

Некоторые СУБД, в том числе Oracle, поддерживают цикл с параметром по курсору;

```
  FOR параметр IN имя_курсора LOOP
    ...
  END LOOP;
```

Использование такого цикла не требует операций открытия и закрытия курсора – они выполняются неявно. Параметр цикла не требуется объявлять в секции DECLARE, его тип определяется автоматически как RECORD, а имена полей записи соответствуют именам в объявлении курсора. Например:

```
FOR cur_rec IN cur LOOP
  ... cur_rec.name_st…
 END LOOP;
```

Из этих объяснений понятно, что использование цикла по курсору – очень простой и удобный способ обработки курсора.

Приведенных сведений уже достаточно, чтобы перейти к практическому применению языка PL/SQL. Его основное назначение – разработка хранимых процедур и функций, а также триггеров базы данных.

4.6.3. Хранимые процедуры и функции

Хранимые процедуры и функции являются стандартными объектами базы данных. Их понимание в SQL не отличается от общепринятого: хранимой подпрограммой (процедурой или функцией) называется именованная, отдельно описанная, повторно используемая программная единица, выполняющая, как правило, определенную прикладную функцию.

Преимущества и недостатки хранимого кода

В современных информационных системах значительная часть бизнес-логики содержится в хранимом коде. Хранимые подпрограммы обеспечивают приложениям баз данных следующие преимущества:

- сокращение объема программирования при разработке приложений, так как однажды созданная подпрограмма может использоваться разными приложениями;
- уменьшение сетевого трафика, так как, если подпрограмма включает в себя несколько обращений к базе данных, по сети передается не каждый запрос и его результат, а только вызов процедуры и ее конечный результат;
- повышение производительности, так как на сервере есть больше возможностей оптимизации локально выполняющихся запросов и здесь могут быть применены средства, недоступные для клиентской части;
- гарантия того, что задача, решаемая хранимой процедурой, будет одинаково выполняться для всех клиентских приложений и для всех клиентских платформ при любых настройках клиентов.

Основным недостатком хранимого кода является его непереносимость между различными СУБД. В силу этого, для масштабируемых ин-

формационных систем или систем, предназначенных для многократного тиражирования, часто используется трехзвенная архитектура системы с переносом значительной части бизнес-логики на уровень сервера приложений. Практикуется и такой вариант: серверная часть тиражируемой информационной системы разрабатывается сразу для нескольких самых распространенных СУБД (например, вариант для Oracle, PostgreSQL и Microsoft SQL Server).

Следует отметить, что хранимые процедуры и функции не следует рассматривать как альтернативу сложным SQL-запросам. По сравнению с «чистым» SQL любая процедурная альтернатива является худшим вариантом с точки зрения производительности, поскольку SQL-запросы любая СУБД обрабатывает с наивысшей эффективностью. Поэтому не нужно поддаваться соблазну и переходить на уровень процедурной логики там, где можно, пусть ценой больших мыслительных усилий, написать сложный, но эффективный запрос с использованием только средств языка SQL.

Создание хранимых процедур и функций

В Oracle традиционно основным языком хранимых процедур является язык PL/SQL, но поддерживаются и процедуры на других языках, прежде всего – на языках C++ и Java. В последнем случае хранимая процедура или функция называется внешней. В рамках нашего курса рассмотрим основной вариант – хранимая процедура на PL/SQL.

Хранимая процедура создается оператором SQL

```
CREATE [OR REPLACE] PROCEDURE имя[(список_параметров)]
AS
блок PL/SQL
```

Необязательная конструкция OR REPLACE позволяет заменять процедуру с таким же именем. Это очень удобно в процессе отладки.

Аналогично создается хранимая функция:

```
CREATE [OR REPLACE] FUNCTION имя[(список_параметров)]
RETURN тип_результата, возвращаемого функцией
AS
блок PL/SQL, обязательно содержащий оператор
RETURN выражение
```

В списке параметров должен быть описан режим использования каждого параметра: IN (только входной – используется по умолчанию), OUT (только выходной), IN OUT (и входной, и выходной). Режим использования указывается после имени параметра. Типы параметров, как и типы переменных, можно указывать явно или с помощью ссылки на соответствующий столбец или таблицу.

При описании локальных переменных подпрограммы разрешено опускать ключевое слово DECLARE.

Удалить хранимую процедуру или функцию можно при помощи команды DROP.

Примеры хранимых процедур и функций

В качестве примеров приведем две хранимые подпрограммы для нашей демонстрационной базы студентов и их оценок.

Первая из процедур демонстрирует применение неявного курсора и предназначена для изменения телефона студента. Ее входными параметрами являются фамилия и новый телефон студента. Конечно, первым входным параметром должна быть не фамилия, а личный код студента, но таким образом мы хотим продемонстрировать исключительную ситуацию TOO_MANY_ROWS. В случае наличия однофамильцев, а также в случае отсутствия студента с такой фамилией процедура должна сообщить о возникшей исключительной ситуации.

Также целесообразно отдельно обработать случай, когда старый телефон совпадает с новым, поэтому операции обновления не требуется.

Для фиксации всех перечисленных выше случаев в процедуру добавлен выходной параметр result, который после завершения процедуры возвращает код ошибки (0 – благополучное завершение процедуры) и может быть обработан клиентским приложением.

```
CREATE PROCEDURE changephone(fiostud students.name_st%TYPE,
                             newphone students.phone%TYPE,
                             result OUT NUMBER)
AS
oldphone students.phone%TYPE; -- старый телефон
BEGIN
SELECT phone  INTO oldphone  FROM students WHERE name_st=fiostud;
IF oldphone!=newphone THEN
UPDATE students SET phone=newphone WHERE  name_st=fiostud;
result:=0;
ELSE
result:=1; -- старый и новый номера совпали
END IF;
EXCEPTION
WHEN NO_DATA_FOUND THEN -- нет такого студента
result:=2;
WHEN TOO_MANY_ROWS THEN
result:=3; -- есть однофамильцы
WHEN OTHERS THEN
result:=4; -- непредвиденная исключительная ситуация
END;
```

Выполнив команду создания данной процедуры в SQL*Plus, мы получим сообщение «Процедура создана». В случае, если в процедуре обнаружены синтаксические ошибки, выдается другое сообщение «Процедура создана с ошибками компиляции». Получить информацию об обнаруженных ошибках можно с помощью запроса к представлению словаря Oracle user_errors:

```
SELECT line, text FROM user_errors
```

Хранимая процедура запускается на выполнение по команде из клиентского приложения.

Чтобы запустить ее на выполнение из SQL*Plus в целях отладки необходимо поместить ее в блок PL/SQL, перед которым объявить переменную для выходного параметра:

```
VAR e NUMBER;
BEGIN
changephone('Иванов', '555555', :e);
END;
```

Проверить значение переменной e можно при помощи команды:

```
PRINT e
```

В качестве второго примера приведем функцию, которая принимает в качестве входного параметра фамилию студента и возвращает строку, содержащую телефоны всех студентов с такой фамилией (возможно, пустую строку, если студентов с такой фамилией нет). Здесь демонстрируется применение явного курсора.

```
CREATE FUNCTION getphone   (fiostud  students.name_st%TYPE)
RETURN  VARCHAR
AS
CURSOR c IS
  SELECT phone FROM students
  WHERE name_st= fiostud; -- телефоны всех студентов с заданной фамилией
  res           VARCHAR(50); -- строка результата
  ph            students.phone%TYPE; -- переменная для команды FETCH
BEGIN
   OPEN c;
   res:='';
   LOOP – цикл для извлечения данных из курсора
      FETCH c INTO ph;
      EXIT WHEN NOT c%FOUND;
      res:=res||ph||'  ';
   END LOOP;
   RETURN res ;
END;
```

Если использовать цикл по курсору, тело функции получится короче:

```
BEGIN
   res:='';
   FOR crec IN c LOOP
   res:=res||crec.phone||'  ';
   END LOOP;
   RETURN res ;
END;
```

После создания функции проверить ее работоспособность можно совсем просто:

```
SELECT getphone('Иванов') FROM dual
```

4.6.4. Триггеры

Триггеры – особый вид хранимых процедур, которые запускаются автоматически при наступлении определенных событий в базе данных.

Особенности триггеров

Являясь по сути хранимой процедурой, триггер обладает теми же преимуществами и недостатками, что и весь хранимый код. К преимуществам следует добавить то, что триггеры являются прекрасным инструментом для администратора БД, поскольку работают независимо от того, какое из клиентских приложений вызвало активизирующее их событие. Эта особенность превращает триггеры также в средство добавления новой функциональности в существующую систему без всякого изменения ее программного кода. Достаточно только выбрать подходящее событие и создать триггер.

Однако, нужно отметить, что использовать триггеры следует с особой осторожностью, ведь клиентские приложения вообще ничего не знают о существовании тех или иных триггеров на сервере, и важно не допустить никаких конфликтов и противоречий в слаженной работе всей информационной системы.

Событий, которые могут активизировать триггеры, довольно много, например, Oracle поддерживает триггеры уровня базы данных, уровня схемы и уровня таблицы. В рамках данного курса рассмотрим только триггеры уровня таблицы, которые обеспечивают автоматическое выполнение некоторых действий при каждой модификации данных таблицы.

Такой триггер характеризуется следующими признаками, которые должны быть заданы при его создании:

- уникальное имя триггера (задание параметров не требуется, поскольку триггер – процедура без параметров);

- активизирующее действие - команда, которая вызывает запуск триггера, такими командами являются `INSERT`, `DELETE`, `UPDATE`;
- время активизации - выполнение триггера до (BEFORE) или после (AFTER) выполнения активизирующего действия;
- область действия - выполнение триггера либо один раз для каждого оператора модификации таблицы, либо для каждой строки (в последнем случае следует добавить фразу FOR EACH ROW);
- условие активизации - необязательное дополнительное условие, которое должно выполняться для запуска триггера (фраза WHEN);
- тело триггера – действия, выполняемые триггером (блок PL/SQL).

На каждое событие может быть создано и несколько триггеров. Однотипные триггеры выполняются в порядке их создания.

Действие, выполняемое в триггере, может включать в себя операции `INSERT`, `DELETE`, `UPDATE`, которые, в свою очередь, могут запускать выполнение того же или других триггеров. Такое явление называется каскадированием триггера.

Команды SQL для работы с триггерами

Триггер создается при помощи команды SQL:

CREATE [OR REPLACE] TRIGGER имя_триггера
время_активизации активизирующая_команда ON имя_таблицы
[FOR EACH ROW]
[WHEN дополнительное условие запуска триггера]
AS
Блок PL/SQL

В теле триггера можно использовать любые операторы PL/SQL, кроме операторов SQL, которые изменяют ту таблицу, для которой был создан данный триггер. Любые другие таблицы изменять можно.

В теле триггера в Oracle можно использовать две предопределенные переменные, которые обозначают ту строку, которая в данный момент подвергается модификации:

:NEW – новое значение строки, применяется для команд INSERT и UPDATE

:OLD – старое значение строки (до модификации), применяется для команд DELETE и UPDATE

Если триггер благополучно создан, далее он будет запускаться сам при любом наступлении активизирующего события. Удалить триггер можно при помощи команды

DROP TRIGGER имя_триггера

Иногда бывают ситуации, когда по каким-либо причинам автоматическое срабатывание триггера не нужно, но и удалять его нельзя, поскольку в дальнейшем он потребуется.

Для временного отключения триггера в Oracle можно применить команду:

ALTER TRIGGER имя_триггера DISABLE

Чтобы снова включить существующий триггер, используют команду:

ALTER TRIGGER имя_триггера ENABLE

Примеры триггеров

1. Триггер на вставку нового студента

При вставке новой строки в таблицу триггеры часто используются для задания таких значений по умолчанию, которые нельзя определить при создании таблицы с помощью фразы DEFAULT. В Oracle триггер на вставку чаще всего используется для автоматического задания значений первичного ключа. В стандарте SQL 2003 для этих целей имеется специальное ключевое слово IDENTITY, но в Oracle оно не поддерживается.

Вместо этого имеется специальный объект **SEQUENCE**, который предназначен для формирования последовательных целых чисел (этот объект зафиксировн в стандарте SQL 2003). Для обращения к значениям последовательности в выражении SQL используются псевдостолбцы CURRVAL и NEXTVAL. CURRVAL возвращает текущее значение. NEXTVAL инкрементирует текущее значение и возвращает результат, при этом он становится текущим значением. Триггер на вставку берет из последовательности очередное значение и помещает его в новую строку, используя предопределенную переменную :NEW.

Например, создадим последовательность для формирования кодов студентов:

```
CREATE SEQUENCE stud_seq
```

Теперь создадим триггер на вставку новой строки в таблицу students:

```
CREATE TRIGGER st_keys
BEFORE INSERT ON students
FOR EACH ROW
BEGIN
SELECT stud_seq.NEXTVAL INTO :NEW.cod_st FROM dual;
END;
```

Аналогичный триггер можно написать и на таблицу subjects, поскольку при добавлении нового предмета его код должен формироваться также автоматически. Для этих целей обычно создают еще одну последовательность, хотя, в принципе и одна последовательность на все таблицы с суррогатными ключами обеспечит уникальность значений ключа в каждой таблице.

2. Триггеры на удаление студента

Триггер на удаление должен предусмотреть перенос удаляемых данных в архивную базу данных. При создании таблиц нашей базы данных вместе с удалением студента было предусмотрено и каскадное удаление оценок студента, поэтому мы можем написать аналогичные триггеры BEFORE DELETE на таблицы students и marks, используя возможность каскадирования триггеров. При удалении студента одна единственная команда удаления, полученная сервером, например:

```
DELETE FROM students WHERE cod_st=125
```

вызовет выполнение двух команд удаления (из таблиц marks и students) вместе с двумя триггерами, сохраняющими удаляемые данные в архиве.

Предположим, что уже созданы таблицы archive_students и archive_marks. Создадим триггер на удаление из таблицы students:

```
CREATE TRIGGER st_del
BEFORE DELETE ON students
FOR EACH ROW
BEGIN
INSERT INTO archive_students
VALUES(:OLD.cod_st, :OLD.name_st, :OLD.born, :OLD.phone);
END;
```

Триггер на удаление из таблицы оценок выглядит аналогично, фраза

```
FOR EACH ROW
```

вызовет его срабатывание при удалении каждой оценки студента.

3. Триггер на изменение оценки

Изменения, вносимые в элементы данных, которые в своей предметной области имеют существенное значение и подлежат усиленному контролю, обычно фиксируются в журналах изменений. В базах данных такие журналы могут представлять собой обычные таблицы и формироваться при помощи триггеров. Триггеры, предназначенные для контроля изменений в важных таблицах, могут быть написаны и на вставку, и на удаление, и на обновление. Однако злоупотреблять этой замечательной возможностью все же не следует, поскольку каждый дополнительный триггер снижает производительность системы.

В нашей демонстрационной базе данных, очевидно, имеет смысл контролировать изменение уже выставленной оценки. Поэтому создадим специальную таблицу change_mark_log (журнал изменений оценок), которая будет содержать столбы:

- name_user (имя пользователя, изменившего оценку)
- date_change (дата изменения оценки)
- cod_st (код студента)
- cod_sub (код предмета)
- old_mark (старая оценка)
- new_mark (новая оценка)

Теперь создадим триггер на обновление:

```
CREATE TRIGGER mark_change
AFTER UPDATE ON marks
FOR EACH ROW
BEGIN
IF :OLD.mark<> :NEW.mark THEN
INSERT INTO change_mark_log
VALUES(user, sysdate, :OLD.cod_st, :OLD.cod_sub, :OLD.mark,
:NEW.mark);
END IF;
END;
```

Проверка работоспособности триггера

Для проверки работоспособности триггера нужно выполнить команду, активизирующую триггер.

Например, для последнего триггера, который заполняет журнал изменения оценок:

```
UPDATE marks SET mark=3 WHERE mark=2
```

Если в таблице оценок были неудовлетворительные оценки, таблица change_mark_log будет содержать исчерпывающие сведения об их изменении. Просмотреть ее администратор базы данных сможет в любое удобное для него время.

Следует отметить, что вопросы создания и удаления триггеров обычно находятся в компетенции администратора базы данных (АБД), а не разработчиков прикладного программного обеспечения. В следующей главе мы кратко коснемся основных проблем, которые необходимо решать АБД в процессе создания и эксплуатации базы данных, и средств, предоставляемых СУБД для этих целей.

5. Управление доступом к данным

Цель изучения данной главы – получить основные знания и умения администрирования базы данных.

После изучения главы вы будете:

- знать основные принципы организации системы безопасности СУБД
- уметь создавать учетные записи пользователей, присваивать им определенные привилегии и роли
- понимать свойства транзакций (АСИД), уметь применять на практике команды SQL для поддержки транзакций
- знать основные принципы управления производительностью СУБД
- уметь создавать необходимые индексы в целях повышения производительности.

5.1. Система безопасности СУБД

Система безопасности (Security) предназначена для сохранности данных от злонамеренной порчи либо от несанкционированного доступа. Обычно система безопасности включает две составляющие:

1. Разграничение доступа пользователей.
2. Аудит действий пользователя.

Кратко рассмотрим функции и реализацию каждой из составляющих. Системы безопасности различных СУБД существенно отличаются друг от друга (что вполне естественно), а стандарт в этой части является достаточно гибким и позволяет строить индивидуальные решения на основе типовых конструкций.

В дальнейшем изложении мы будем опираться на систему безопасности Oracle, понимая, что при переходе на любую новую СУБД придется потратить определенное время на освоение ее системы безопасности.

5.1.1. Разграничение доступа пользователей

Создание учетных записей пользователей

Все СУБД предоставляют доступ к информации базы данных только зарегистрированным пользователям, имеющим учетные записи этой базы. При установке Oracle автоматически создается несколько учетных записей – это пользователи с именами SYS и SYSTEM, обладающие всеми правами АБД (самыми обширными правами обладает SYS, который имеет право останавливать и запускать сервер), а также несколько пользователей, созданных в демонстрационных целях.

Для создания учетной записи каждого нового пользователя требуется создать стандартный объект USER с обязательным указанием пароля данного пользователя (пароли в словаре данных Oracle хранятся в зашифрованном виде, поэтому если пользователь забыл пароль, его нельзя посмотреть, но можно сменить). В Oracle команда CREATE USER … содержит еще ряд дополнительных параметров:

CREATE USER имя_пользователя
IDENTIFIED BY пароль
[DEFAULT TABLESPACE имя_табличного_пространства]
[TEMPORARY TABLESPACE
имя_временного_табличного_пространства]
[QUOTA UNLIMITED / размер предоставляемого дискового пространства]
[PROFILE имя_профиля]

Например:

```
CREATE USER user1
IDENTIFIED BY cneltyn
DEFAULT TABLESPACE USERS
TEMPORARY TABLESPACE TEMP
```

В данной команде используются стандартные имена табличных пространств, которые создаются при установке сервера Oracle. Табличное пространство – это файл (или несколько файлов, логически воспринимаемые Oracle как единое пространство для хранения таблиц или индексов). В приведенном примере размер части табличного пространства, предоставляемого пользователю, не ограничен (точнее, ограничен только размерами пространства). Специальный профиль для пользователя не создается.

Несколько слов о профилях пользователей (подробные сведения выходят за рамки курса). Создание индивидуальных профилей для пользователей базы данных позволяет:

1. Вести индивидуальную политику паролей (время жизни пароля)
2. Ограничить ресурсы (количество одновременно открытых сессий, процессорное время, отводимое данному пользователю, или время для некоторого конкретного запроса, время ожидания ответа сервера и ряд других важных ресурсов).

Профиль создается при помощи команды

CREATE PROFILE имя профиля [параметры_профиля]
и в любой момент может быть изменен командой
ALTER PROFILE …..

Изменение учетной записи

Команда **ALTER USER имя ......** позволяет изменять различные параметры учетной записи (кроме имени пользователя).

Например:

ALTER USER user1
IDENTIFIED BY новый_пароль

Так можно сменить пароль пользователя.

ALTER USER user1 QUOTA 100M ON USERS

Теперь для пользователя user1 размер его части табличного пространства USERS ограничен 100 Мегабайтами.

Удаление учетной записи

С помощью команды **DROP USER...** можно совсем удалить пользователя из базы данных. Например, команда:

DROP USER user1

удалит пользователя user1, но только в том случае, если он еще не создал ни одного объекта.

Для того, чтобы гарантированно удалить любого пользователя, требуется применить эту команду с параметром CASCADE, с помощью которого перед удалением пользователя удаляются все объекты в его схеме.

DROP USER user1 CASCADE

Пользователи и схемы

При создании учетной записи в Oracle каждый пользователь одновременно получает в распоряжение свою собственную схему в базе данных с тем же именем, т.е одновременно неявно выполняется команда

CREATE SCHEMA имя_пользователя

Напомним, что понятие схемы поддерживается стандартом. Это самостоятельная часть базы данных, все имена объектов, создаваемых пользователем, должны быть уникальны только в пределах схемы, поскольку имя объекта (таблицы, представления и т.д.) в пределах всей базы данных складывается из двух составляющих:

имя_схемы.имя_объекта

Однако, получив в распоряжение собственную учетную запись и собственную схему базы данных, только что созданный пользователь не может даже подключиться к базе данных, не говоря уже о создании объектов. Для выполнения любого действия в базе данных пользователю требуются права на выполнение подобных действий. Для этих целей вводятся два понятия – привилегии и роли.

5.1.2. Привилегии и роли

Привилегии позволяют использовать определенные команды языка SQL по отношению к определенным ее объектам и предоставляются с помощью команды **GRANT**… Например, если пользователь user1 владеет таблицей students и выполняет команду

GRANT SELECT ON students TO PUBLIC

все пользователи (PUBLIC) смогут выполнять команду SELECT по отношению к таблице students, используя для этого составное имя user1.students или короткий синоним, если он будет создан при помощи команды:

CREATE PUBLIC SYNONYM students FOR user1.students

Из этого примера очевидно, что для нормальной работы большинству пользователей необходимо предоставить очень большое количество привилегий.

Упрощают работу с привилегиями роли – именованные группы привилегий, которые можно предоставлять пользователям с помощью той же команды GRANT, что и отдельные привилегии. В приложениях с большим числом пользователей применение ролей намного уменьшает количество команд GRANT. Можно создать заранее определенный набор ролей, с помощью которых пользователям предоставляются только те привилегии, которые им необходимы.

Роль – это стандартный объект базы данных, поэтому она создается при помощи стандартной команды CREATE…

CREATE ROLE имя_роли

Например, создадим роль student, которую могут получить все пользователи-студенты, изучающие курс «Базы данных».

CREATE ROLE student

Точно так же, как и только что созданный пользователь, новая роль еще не содержит никаких привилегий, поэтому присваивать эту роль пользователям бессмысленно.

Наполнить роль набором конкретных привилегий в Oracle можно при помощи уже упоминавшейся команды GRANT. Теперь уже можно представить эту команду в общем виде:

GRANT список_привилегий/ролей TO
список_пользователей/ролей/PUBLIC

Отменить привилегию или роль, которые присвоены командой GRANT, можно с помощью другой команды:

REVOKE список_привилегий/ролей FROM список_пользователей/ролей/PUBLIC

Как видим, команды GRANT и REVOKE являются универсальными, а сама система присвоения и отмены привилегий – очень гибкой. Новой

роли можно присвоить как конкретные привилегии, так и другие роли. Например, команда:
GRANT connect, resource TO student
присвоит роли student две стандартные роли, которые имеются во всех версиях Oracle – роль connect позволяет подключаться к базе данных, а роль resource позволяет создавать некоторые объекты в своей схеме (таблицы, индексы, представления, хранимый код). Если в целях успешного обучения пользователя с ролью student ему необходимо предоставить все права администратора, это можно сделать при помощи команды:
GRANT dba TO student

Такую роль уже не стыдно присвоить успешному студенту, допустим, ранее созданному при помощи команды CREATE USER user1 ..
GRANT student TO user1

Однако в реально функционирующих информационных системах использование стандартных ролей Oracle часто нежелательно, поскольку некоторые пользователи могут получить избыточные привилегии, при этом не получить какие-то нужные им специфические права. Поэтому проанализируем множество существующих привилегий, чтобы понять логику их присвоения пользователям и ролям.

Как известно, основными составными частями языка SQL являются DDL и DML. В соответствии с этим выделяют две больших группы привилегий – системные и объектные.

Системные привилегии

Системные привилегии получают администраторы БД и, частично, разработчики приложений. Если разработчикам приложений дается в распоряжение собственная тестовая база данных, они получают на нее любые системные привилегии, которые им необходимы в процессе разработки и тестирования приложений.

Системные привилегии – это права на выполнение команд CREATE, ALTER DROP применительно к различным объектам базы данных в своей (или любой) схеме. Если пользователи или роли наделяются привилегией на выполнение DDL в любой схеме, к имени объекта добавляется клюсевое слово ANY. Например:
GRANT CREATE TABLE TO student
позволит включить в роль студент право создавать таблицы в своей схеме, а
GRANT CREATE ANY TABLE TO student WITH GRANT OPTION
право создавать таблицы в любой схеме, при этом роль student сможет передать эту привилегию другим пользователям или ролям.

Важно отметить, что пользователь, создавший таблицу (неважно где – в своей или чужой схеме), автоматически становится ее *владельцем*

(owner), при этом он автоматически получает все права объектного уровня на эту таблицу. Другими словами, владельцу таблицы не надо предоставлять отдельные права на манипулирование данными этой таблицы, но всем остальным пользователям такие права предоставлять нужно.

Аналогично можно присвоить привилегии на любые другие команды DDL. Особо требуется отметить привилегию CREATE SESSION, которую должны получить все пользователи без исключения, поскольку это право на открытие сеанса связи с сервером Oracle (дословно – создание сессии).

Привилегия ALTER SESSION, скорее всего, не потребуется рядовым пользователям, поскольку предоставляет право изменять важные системные параметры в своем сеансе связи, которые могут повлиять, например, на производительность системы.

В качестве примера создадим системную роль account_creator, с помощью которой можно только создавать пользователей, а другие команды уровня DBA выполнять нельзя:

```
CREATE ROLE account_creator
GRANT CREATE SESSION, CREATE USER, ALTER USER
TO account_creator
```

С помощью первой команды создается роль с именем account_creator. Вторая команда предоставляет этой роли возможность регистрации в базе данных, а также создания и изменения учетных записей.

Объектные привилегии

Объектные привилегии позволяют пользователям выполнять конкретные действия на конкретном объекте. Такова, например, привилегия удалять строки в указанной таблице. Объектные привилегии назначаются конечным пользователям, так что они могут использовать приложения базы данных для выполнения конкретных задач.

Объектные привилегии выдаются при помощи уже известной универсальной команды GRANT…

GRANT список_объектных_привилегий
ON имя таблицы/представления/процедуры
TO список_ролей/список_пользователей/PUBLIC
[WITH GRANT OPTION]

В ORACLE существуют следующие привилегии объектного уровня:

- SELECT позволяет другому пользователю выполнить запрос на выборку к данным указанной таблицы или представления.
- INSERT позволяет вставлять строки в таблицу (возможно, используя для этих целей обновляемое представление) с помощью команды INSERT.

- UPDATE позволяет обновлять строки в таблице (обновляемом представлении) вне зависимости от того, были ли эти строки созданы этим пользователем или нет.
- DELETE позволяет удалять из таблицы любые существующие строки. С использованием представления можно ограничить то, какие строки будут удалены.
- EXECUTE даёт возможность пользователю, владеющему хранимым кодом базы данных (процедурами, функциями или пакетами), позволить другому пользователю ORACLE вызывать его процедурные объекты.
- ALTER даёт возможность пользователю ORACLE изменить определение заданной таблицы или последовательности. Не следует путать с системной привилегией ALTER TABLE, которая дает возможность изменять структуру любой таблицы в своей схеме.
- INDEX позволяет пользователю создавать индексы на указанную таблицу, владельцем которой он не является. Владелец эту привилегию имеет по умолчанию.

В конце оператора GRANT привилегии объектного уровня может быть определена фраза WITH GRANT OPTION, которая позволяет пользователю, получившему эту привилегию, передать её другому пользователю ORACLE.

Например, пусть ранее был создан пользователь user2. Предоставим ему только некоторые права пользователя нашей демонстрационной базы данных:

```
GRANT CREATE SESSION TO user2
GRANT SELECT, INSERT ON marks TO user2
GRANT SELECT ON students TO user2
GRANT EXECUTE ON change_phone TO user2
```

5.1.3. Аудит действий пользователей

Разграничение прав доступа пользователей вовсе не отменяет необходимость контроля (аудита) их действий в базе данных. Ранее мы рассмотрели, как организовать аудит некоторых действий при помощи триггеров.

Однако Oracle предоставляет дополнительнительную возможность всестороннего аудита всех операций, происходящих в базе данных, - команду **AUDIT**. Информация аудита записывается либо в специальную таблицу в схеме SYS (SYS.AUD$), либо (в зависимости от применяемой системы) в журнал аудита операционной системы.

Разрешается проводить аудит операций трех разных типов:

- попыток регистрации в базе данных (аудит подключений),

- обращения к определенным объектам (аудит объектов),
- определенных операций с базой данных (аудит действий).

Во время аудита база данных по умолчанию должна регистрировать как успешные, так и неуспешные команды. Этот режим можно изменить при установке аудита любого типа.

Аудит подключений

Можно осуществлять аудит всех попыток соединения с базой данных. Аудит попыток регистрации задается командой

AUDIT SESSION

Чтобы производить аудит только тех попыток регистрации, которые завершаются или успешно или неуспешно, воспользуйтесь одной из следующих команд:

AUDIT SESSION WHENEVER SUCCESSFUL

AUDIT SESSION WHENEVER NOT SUCCESSFUL

Если записи аудита хранятся в таблице SYS.AUD$, их можно просмотреть через представление словаря данных DBA_AUDIT_SESSION этой таблицы.

Для запрещения аудита сеанса применяется команда NOAUDIT

NOAUDIT SESSION

Аудит операций

Любая команда DDL, оказывающая воздействие на некоторый объект базы данных, например на таблицу, представление или индекс, может быть подвергнута аудиту. При этом нетрудно сгруппировать операции CREATE, ALTER и DROP, воздействующие на объекты. Группирование команд снижает объем административной работы, необходимой для установки и поддержки параметров аудита.

Так, для аудита всех команд, воздействующих на роли, нужно ввести команду

AUDIT ROLE

Чтобы отменить заданную установку, следует ввести

NOAUDIT ROLE

Существуют группы аудита SQL-команд. Каждую группу можно применять для аудита всех SQL-команд, входящих в нее. Например, с помощью команды AUDIT ROLE, приведенной выше, будет осуществляться аудит команд CREATE ROLE, ALTER ROLE и DROP ROLE.

Также можно осуществлять отдельный аудит для каждой команды, указанной операторным параметром. ORACLE предлагает следующие группы операторных параметров:

CONNECT	аудит всех случаев регистрации в ORACLE и отсоединения от ORACLE
DBA	аудит команд, для выполнения которых необходимы полномочия администратора базы данных
RE-SOURCE	аудит операций создания и удаления таблиц, кластеров, представлений, индексов, табличных областей, типов и синонимов
ALL	аудит всех этих команд

Аудит объектов

Помимо системных операций, выполняемых над объектами, аудиту можно подвергать операции SELECT, INSERT, UPDATE и DELETE, выполняемые над конкретными таблицами.

Конструкция, добавляемая для аудита объектов, - BY SESSION (на сеанс) или BY ACCESS (по доступу). Она определяет, нужно ли вносить запись аудита однажды для каждого сеанса или каждый раз при обращении к объекту. Например, если пользователь выполнил над одной и той же таблицей четыре различных оператора UPDATE, результатом аудита по доступу будет внесение четырех записей аудита – по одной на каждое обращение к таблице. Однако если в той же самой ситуации применить конструкцию BY SESSION, то будет внесена только одна запись аудита.

Поэтому аудит по доступу может намного увеличить частоту внесения записей аудита. Он используется достаточно редко и, как правило, для измерения числа отдельных операций, выполняемых в течение определенного временного интервала; после завершения тестирования следует установить для аудита состояние BY SESSION.

Ниже рассмотрены примеры использования рассмотренных способов аудита. В первой команде производится аудит всех команд INSERT, выполняемых над таблицей students, находящейся в схеме user1. Во второй команде аудиту подвергается каждая команда, воздействующая на таблицу marks. В третьей команде осуществляется аудит операций DELETE, выполняемых над таблицей subjects в течение сеанса:

```
AUDIT INSERT ON use1.students
AUDIT ALL ON user1.marks
AUDIT DELETE ON user1.subjects BY SESSION
```

5.2. Поддержка транзакций

Транзакция – единица работы СУБД, которая может быть выполнена либо целиком, либо вообще не выполнена. Объем транзакции может варьироваться от одного SQL-оператора до всех действий с базой данных, выполняемых приложением. Чтобы понять суть механизма транзакций, рассмотрим основные свойства, характеризующие транзакцию.

5.2.1. Свойства транзакции

Транзакция характеризуется четырьмя основными свойствами, часто называемыми свойствами АСИД – Атомарность, Согласованность, Изолированность, Долговечность. На английском языке эта аббревиатура также обозначается ACID - Atomicity, Consistency, Isolation, Durability. Поясним каждое из свойств по отдельности.

Атомарность

Транзакция является неделимой, она выполняется полностью или не выполняется вообще; если транзакция прерывается на середине, то база данных должна остаться в том состоянии, которое она имела до начала транзакции.

Согласованность (целостность)

Транзакция переводит базу данных из одного согласованного (целостного) состояния в другое, также целостное. В ходе выполнения транзакции база данных может временно пребывать в нецелостном состоянии.

Очень многие правила целостности базы данных таковы, что их просто невозможно не нарушить, выполнив только одну команду SQL. Такие команды объединяют в единую транзакцию, результатом которой является новое целостное состояние БД. В крайнем случае, если транзакция не сумеет довести БД до конечного целостного состояния, она вернется к начальному, также целостному, состоянию.

Поясним это свойство на примере. Рассмотрим процесс удаления студента из нашей демонстрационной базы данных с перенесением данных о студенте и всех его оценках в архивную базу данных. В предыдущей лекции были рассмотрены триггеры для выполнения переноса в архив всех строк таблицы оценок marks, относящихся к данному студенту, и строки с данными о самом студенте. Схематически процесс удаления строки из таблицы students со всеми сопутствующими действиями показан на рис.5.1.



Рис. 5.1. Процесс удаления строки из таблицы students

Он состоит из четырех команд SQL, выполняемых последовательно, причем все промежуточные состояния базы данных после выполнения отдельных команд являются несогласованными. При возникновении любой внештатной ситуации (допустим, пользователь не имеет привилегий на выполнение какой-либо одной из операций или на архивные таблицы наложены какие-либо дополнительные ограничения) будет *автоматически* выполнен откат в исходное состояние.

Наконец, после выполнения последнего удаления система снова окажется в согласованном состоянии. Но и в этот момент еще возможен откат в исходное состояние, на этот раз по команде ROLLBACK, которая уже инициируется пользователем, а не сервером. Откат станет невозможным только после выполнения команды фиксации транзакции COMMIT. Любая из этих двух команд, тем или иным образом, но завершает транзакцию.

Изоляция

До сих пор мы не рассматривали особенности работы СУБД в многопользовательской среде. Но в реальных информационных системах на основе сервера Oracle могут одновременно работать десятки тысяч пользователей, каждый из которых выполняет свои собственные транзакции. Свойство изоляции обозначает, что *все транзакции выполняются изолированно друг от друга.*

Несмотря на то, что в каждый момент времени сервер может выполнять параллельно несколько транзакций, общий результат их совместной работы гарантировнно будет таким же, каким он был бы при их последовательном выполнении; одновременно выполняющиеся транзакции не наложатся и не исказят результаты друг друга.

Такой режим работы сервера называется сериальным и является режимом по умолчанию для сервера Oracle.

Долговременность

После того как транзакция завершена и зафиксирована, результат ее выполнения гарантированно сохраняется в базе данных. При любых аварийных ситуациях, используя возможности сервера, можно восстановить *все зафиксированные транзакции.* Восстановление незафиксированных транзакций сервер, разумеется, не гарантирует.

5.2.2. Поддержка транзакций в языке SQL

Для обеспечения всех перечисленных выше свойств транзакции необходима как языковая поддержка процессов инициализации и завершения транзакции, так и механизмы их реализации сервером. Рассмотрим данные вопросы по порядку.

Языковые правила поддержки транзакций для различных СУБД несколько различаются. Например, в Microsoft SQL Server поддерживается команда начала транзакции BEGIN TRANS[ACTION]. В Oracle такой команды нет, но существуют четкие правила, регламентирующие моменты начала и завершения транзакции:

1. Любая команда DDL выполняется как отдельная транзакция. Иными словами, поступлснис на ссрвср команды DDL автоматичсски фиксирует результаты предыдущих команд DML этого сеанса (если таковые были) и начинает новую транзакцию, а при завершении команды DDL автоматически фиксируются ее результаты. Таким образом, одна команда DDL вызывает те же действия, что и последовательность команд:
COMMIT
команда DDL
COMMIT

Получается, что несколько команд DDL нельзя объединить в единую транзакцию и откатить команду DDL при помощи стандартной команды ROLLBACK в Oracle также нельзя.

2. Результаты выполнения команд DML автоматически фиксируются только при включенном режиме AUTOCOMMIT (например, в настройках утилиты SQL*Plus есть возможность включения этого режима). По умолчанию этот режим отключен. Таким образом, все идущие подряд команды DML воспринимаются как одна транзакция.

Инициализация транзакции (неявная команда BEGIN TRANSACTION) происходит в следующих случаях:

- первая команда в сеансе связи
- первая команда после команд COMMIT или ROLLBACK
- первая команда после команды DDL

Завершение транзакции происходит при поступлении команд COMMIT (завершение транзакции с фиксацией изменений) или ROLLBACK (завершение транзакции с откатом изменений). Можно неявно зафиксировать команду транзакции из последовательности команд DML любой следующей за ней командой DDL.

Стандарт SQL и многие СУБД, в том числе Oracle, предусматривают так называемые точки сохранения. Точка сохранения задается оператором

`SAVEPOINT` имя_точки_сохранения

и в операторе `ROLLBACK` имеется возможность отката транзакции не к началу, а к указанной точке сохранения:

`ROLLBACK TO` имя_точки_сохранения

Данная команда выполняет откат только тех изменений, которые были сделаны после точки сохранения, и *не завершает транзакцию*.

5.2.3. Механизмы СУБД для поддержки транзакций

Поддержка транзакций требует значительных ресурсов и существенно (во много раз!) замедляет производительность сервера. Однако в современных условиях допустить потерю или порчу информации в базе данных абсолютно недопустимо, поэтому правила АСИД реализуются всеми СУБД. Несмотря на особенности конкретных реализаций, имеется ряд универсальных механизмов поддержки транзакций.

Мы рассмотрим эти обшие механизмы, добавив немного конкретных сведений по их воплощению в Oracle.

Журнализация транзакций

Ведение журналов транзакций преследует одновременно две цели:

1) возможность отката транзакции;
2) восстановление БД в случае аварийных ситуаций или сбоев.

Сервер ведёт 2 вида журналов транзакций:

Undo-журналы:
используются для отката и ведутся для каждой транзакции отдельно. Как только очередная транзакция зафиксирована или откачена, то информация из соответствующего Undo-журнала удаляется.

В Oracle такие журналы иначе называются сегментами отката, они располагаются в отдельном табличном пространстве и содержат полную информацию о каждой транзакции, достаточную для ее успешного отката. Рекомендуемое количество Undo журналов в базе данных, которые должны быть в наличии, равно n/2 (где n – количество одновременно работающих пользователей).

Redo-журнал:
необходим для повторного выполнения транзакций при восстановлении данных. Это единый системный журнал, в который записываются результаты всех зафиксированных транзакций.

В аварийной ситуации, приведшей к порче данных, они восстанавливаются по резервной копии. Но резервная копия была снята в какой-то момент времени в прошлом. Все транзакции, которые прошли в системе после снятия последней резервной копии, восстанавливаются по Redo-журналу.

Такой механизм полностью гарантирует выполнение правила Долговечность из аббревиатуры АСИД. Для того чтобы гарантировать возможность безусловного восстановления всех зафиксированных транзакций, принято правило упреждающей записи в журнал транзакций – по команде COMMIT результаты транзакции сначала заносятся в Redo-журнал, а потом (возможно не сразу), записываются в табличное пространство.

Кратко рассмотрим, как организовано ведение системного журнала транзакций в Oracle. С учетом огромного количества информации, которая постоянно записывается в этот журнал в процессе нормальной работы информационной системы, он состоит из двух частей – оперативной и архивной. Оперативная часть, в свою очередь, содержит два журнала (для повышения надежности каждый из журналов обычно ведется в двух экземплярах). В каждый момент времени один из журналов заполняется информацией о транзакциях сервера, а другой в это время копируется в архивный журнал, который организован на отдельном носителе большого объема. Этот процесс схематически показан на рис. 5.2.

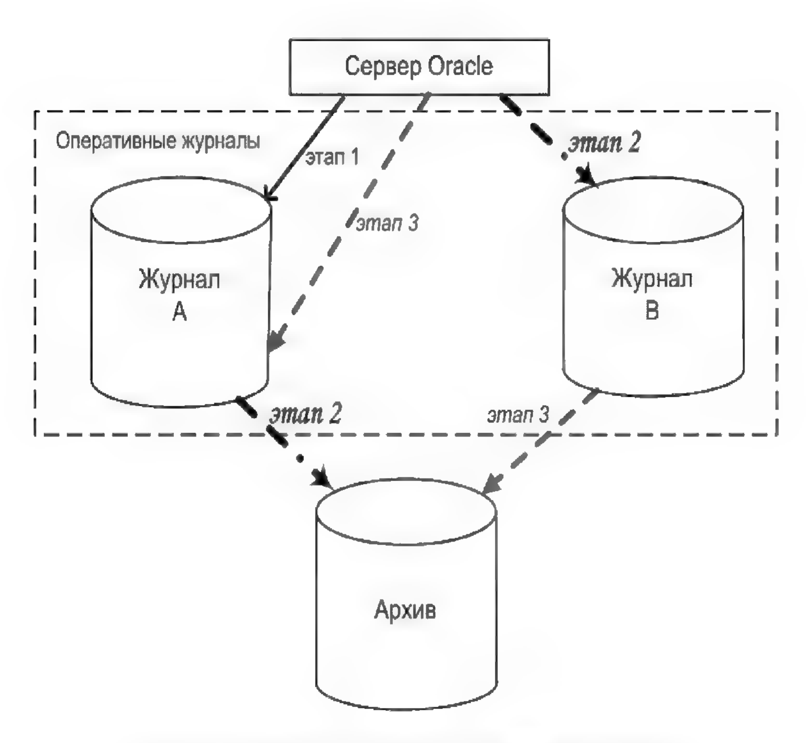


Рис. 5.2. Заполнение журнала транзакций в Oracle

Сериальный график исполнения транзакций. Монитор транзакций

Для обеспечения свойства изолированности транзакций при максимально возможной в этих условиях производительности сервера используется механизм сериализации транзакций. Сериальный график исполнения транзакций обеспечивает параллельное исполнение сервером нескольких транзакций (т.е. команды различных транзакции могут исполняться на сервере «вперемешку»), но результат работы гарантированно должен быть точно таким же, как если бы эти транзакции исполнялись последовательно.

Составлением графика выполнения транзакций, пришедших на сервер, занимается специальный компонент СУБД, который называется *монитором транзакций.* Это очень сложный программный модуль, выполняющий задачу оптимизации суммарного времени исполнения транзакций с учетом блокировок ресурсов в процессе одновременного обращения к ним нескольких транзакций.

Механизмы блокировки ресурсов рассмотрим отдельно.

Механизмы блокировки

Большинство СУБД позволяют любому числу транзакций одновременно осуществлять доступ к одной и той же базе данных и в них существуют механизмы управления параллельными процессами, предотвращающие нежелательные воздействия одних транзакций на другие. По сути, это механизм блокирования, главная идея которого достаточно проста. Если транзакции нужны гарантии, что некоторый объект (база данных, таблица, или строка), в котором она заинтересована, не будет изменен каким-либо непредсказуемым образом в течение требуемого промежутка времени, она устанавливает блокировку этого объекта. Результат блокировки заключается в том, чтобы изолировать этот объект от других транзакций и предотвратить его изменение средствами этих транзакций.

Рассмотрим кратко, как организованы блокировки в Oracle.

Во-первых, объектом блокирования может быть вся таблица или отдельная строка, поэтому различают блокировки типа T (Table) или R (Row). Объектом блокирования в исключительных случаях может быть и вся база данных, но при обычной многопользовательской работе этого не допускают.

Во-вторых, поддерживается не только полная блокировка ресурса (эксклюзивная блокировка – обозначается как X-блокировка), но и разделяемая (Shared) блокировка (обозначается как S-блокировка). Разделяемая блокировка позволяет нескольким транзакциям одновременно использовать ресурс, но пока не снята разделяемая блокировка, сервер не может наложить на этот ресурс эксклюзивную блокировку.

Теперь разберем конкретно, какую блокировку накладывает сервер на ресурсы при исполнении определенных команд SQL.

Команды CREATE/ALTER/DROP TABLE накладывают эксклюзивную блокировку на уровне таблицы (*блокировка TX*). Это значит, что нельзя выполнять никаких действий над таблицей, пока не будет закончена соответствующая операция DDL. Очевидно, для команд DDL и фиксация выполняет автоматически с целью быстрее освободить таблицу от эксклюзивной блокироки.

Команды INSERT/DELETE/UPDATE используют ресурсы в более мягком режиме. Каждая из этих команд накладывает эксклюзивную блокировку только на ту строку, которую она в данный момент обрабатывает (*блокировка RX*). Однако одновременно накладывается эксклюзивная блокировка на всю таблицу (*блокировка TS*), и это означает, что никакая DDL операция не может быть выполнена над таблицей до тех пор, пока не закончатся все DML-операции над этой таблицей и не будет снята последняя TS-блокировка (при нормальной работе такое случится, скорее всего, только к концу рабочего дня или в обеденный перерыв).

Команда SELECT не накладывает никакой блокировки на те таблицы, данные из которых она выбирает. Сервер Oracle гарантирует каждой команде выборки неизменность данных таблиц в процессе ее выполнения. Если в процессе выполнения какой-нибудь объемной и протяженной по времени выборки успела зафиксироваться какая-либо транзакция обновления данных, команда выборки не увидит результаты этой транзакции.

5.3. Настройка производительности. Индексы

Управление большими и сверхбольшими базами данных, проектирование и разработка приложений для них имеет свои особенности. Поэтому, чтобы не допустить ухудшения характеристик как отдельных приложений, так и всей системы в целом, требуется использование специальных методов, повышающих скорость доступа к данным.

Как правило, используется комплексный подход. Под этим понимается оптимизация всех звеньев системы — серверной, клиентской и сетевой части. Существует целый ряд способов настройки производительности: настройка рабочих станций клиентов, сетевого транспорта, оптимизация клиентских приложений, оптимизация серверного PL/SQL-кода и SQL-запросов. В данной лекции мы подробно рассмотрим основной способ повышения производительности при исполнении SQL-запросов – использование индексов.

5.3.1. Понятие индекса

Прежде чем рассматривать различные способы организации индексов, следует ввести понятие *селективности столбца.*

Селективность (*selectivity*) столбца — это процент строк, имеющих одинаковое значение для индексированного столбца. Селективность столбца высокая, если в нем мало одинаковых значений. Автоматически создаются индексы для первичных ключей или столбцов, для которых существует ограничение на уникальность значений. Эти индексы наиболее эффективны. Столбцы с малым количеством уникальных значений имеют низкую селективность, большинство распространенных способов организации индексов при поиске по столбцам с низкой селективностью не обеспечивают существенного ускорения.

Наряду с селективностью столбца при решении вопроса о целесообразности индексирования по тому или иному столбцу требуется учитывать динамику обновления данных в этом столбце (и вообще в таблице). Дело в том, что в процессе изменения данных столбца приходится вносить изменения не только в таблицу, но и в индекс на основе данного столбца, что замедляет операции обновления данных.

Таким образом, *целесообразно создавать индексы для столбцов с высокой селективностью, для которых чаще выполняются операции поиска данных, чем их обновления.*

Нужно отметить важный факт — если в запросах на выборку логическое условие накладывается не на значение столбца, а на результат некоторой функции от него, то индекс, созданный для столбца, не используется.

Индексы бывают двух видов — простые и составные. Составной индекс — это индекс, включающий более чем один столбец. Можно совместно проиндексировать два или более столбца, каждый из которых обладает низкой селективностью, а пара их значений — высокой. Если все столбцы, используемые запросом, входят в составной индекс, то обращения к таблицам можно вовсе избежать — все данные будут считаны только из индекса. Для эффективного использования составного индекса необходимо, чтобы логические условия были наложены на ведущие столбцы индекса, то есть на те столбцы, которые были указаны первыми при создании индекса.

Как правило, в индексах хранятся значения индексируемых столбцов таблицы и физические адреса строк для каждого из хранимых значений столбца (столбцов). В Oracle у каждой строки таблицы есть собственный уникальный идентификатор *ROWID,* который полностью определяет ее физический адрес на диске. Чтобы найти строку в таблице по заданному значению столбца, необходимо найти соответствующие *ROWID* в индексе и затем сразу перейти к указанным ими строкам в таблице.

Как уже упоминалось, индексы являются вспомогательным объектом, поэтому они не поддерживаются стандартом SQL. Тем не менее, все СУБД используют индексы и больших принципиальных различий в политике использования индексов не наблюдается.

На сегодняшний день создание и поддержка индексов в большинстве случаев является обязанностью администратора базы данных. Если разработчик имеет привилегию создания индексов, он самостоятельно может создавать индексы в целях ускорения своих поисковых запросов. Чтобы принимать взвешенное решение о создании тех или иных индексов, нужно хорошо знать, какие именно способы организации индексов поддерживает конкретная СУБД.

Далее мы остановимся на основных способах индексирования, принятых в Oracle.

5.3.2. Обзор индексов Oracle

В Oracle имеется несколько типов индексов:

- древовидные индексы (B-деревья).
- хешированные индексы (*hash*).
- индексы на основе битовых карт или битовые индексы (*bitmap*).

B-деревья были реализованы в Oracle практически с самого начала ее существования, затем появились хешированные индексы, а затем - битовые карты.

Понимание того, когда и где следует использовать конкретные типы индексов, очень важно для эффективного их применения. B-деревья используются наиболее часто, в то время как хешированные и битовые индексы лишь при наличии некоторых условий могут обеспечить существенные преимущества в выполнении определенных запросов.

Оператор создания индекса использует следующий синтаксис:

CREATE [UNIQUE| BITMAP] INDEX имя_индекса
ON имя_таблицы (имя_столбца , [...])

Для удаления индекса используется команда

DROP INDEX <ИМЯ> (удалить)

Можно перестроить существующий индекс без его удаления и повторного создания при помощи команды:

ALTER INDEX<ИМЯ> REBUILD (перестроить индекс)
ALTER INDEX<ИМЯ> UNUSABLE (отключить индекс на время, чтобы снова включить обратно при помощи REBUILD)

Далее рассматривается, как работают индексы, а также приводятся рекомендации, в каких случаях и почему их следует использовать.

В-деревья

Видимо, наиболее популярным подходом к организации индексов в базах данных является использование техники B-деревьев. B-дерево содержит по одному индексному элементу для каждой строки таблицы, в которой имеется непустое (NOT NULL) индексное значение. С точки зрения внешнего логического представления, B-дерево - это сбалансированное сильно ветвистое дерево во внешней памяти (рис.5.3).

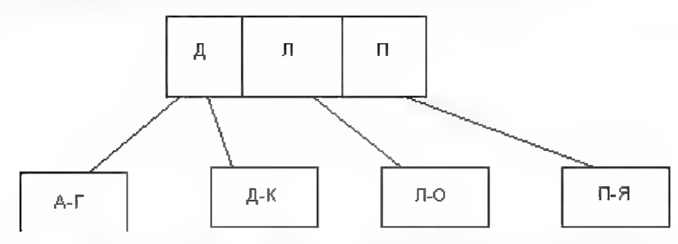


Рис. 5.3. Древовидный индекс по текстовому столбцу

С точки зрения физической организации, В-дерево представляется как мультисписочная структура страниц внешней памяти, т.е. каждому узлу дерева соответствует блок внешней памяти (страница). Внутренние и листовые страницы обычно имеют разную структуру.

В типовом случае структура внутренней страницы выглядит следующим образом:

N1 ключ(1) N2 ключ(2) N3 .... Nn ключ(n) N(n+1) ключ(n+1)

При этом выдерживаются следующие свойства:
ключ(1) <= ключ(2) <= ... <= ключ(n);
в странице дерева Nm находятся ключи k со значениями ключ(m) <= k <= ключ(m+1).

Листовая страница обычно содержит значение индекса и идентификаторы строк (ROWID) и имеет следующую структуру:

ключ(1) сп(1) ключ(2) сп(2) .... ключ(t) сп(t)

Листовая страница обладает следующими свойствами:

- ключ(1) < ключ(2) < ... < ключ(t);
- сп(r) - упорядоченный список идентификаторов кортежей (tid), включающих значение ключ(r);
- листовые страницы связаны одно- или двунаправленным списком.

Поиск в В-дереве - это прохождение от корня к листу в соответствии с заданным значением ключа. Заметим, что поскольку деревья сильно ветвистые и сбалансированные, то для выполнения поиска по любому значению ключа потребуется одно и то же (и обычно небольшое) число обменов с внешней памятью. Более точно, в сбалансированном дереве, где длины всех путей от корня к листу одни и те же, если во внутренней странице помещается n ключей, то при хранении m записей требуется дерево глубиной $\log_n(m)$. Если n достаточно велико (обычный случай), то глубина дерева невелика, и производится быстрый поиск.

Основной "изюминкой" В-деревьев является автоматическое поддержание свойства сбалансированности. Рассмотрим, как это делается при выполнении операций занесения и удаления записей.

При занесении новой записи выполняется:

- Поиск листовой страницы. Фактически, производится обычный поиск по ключу. Если в В-дереве не содержится ключ с заданным зна-

чением, то будет получен номер страницы, в которой ему надлежит содержаться, и соответствующие координаты внутри страницы.

- Помещение записи на место. Естественно, что вся работа производится в буферах оперативной памяти. Листовая страница, в которую требуется занести запись, считывается в буфер, и в нем выполняется операция вставки. Размер буфера должен превышать размер страницы внешней памяти.
- Если после выполнения вставки новой записи размер используемой части буфера не превосходит размера страницы, то на этом выполнение операции занесения записи заканчивается. Буфер может быть немедленно вытолкнут во внешнюю память или временно сохранен в оперативной памяти в зависимости от политики управления буферами.
- Если же возникло переполнение буфера (т.е. размер его используемой части превосходит размер страницы), то выполняется расщепление страницы. Для этого запрашивается новая страница внешней памяти, используемая часть буфера разбивается, грубо говоря, пополам (так, чтобы вторая половина также начиналась с ключа), и вторая половина записывается во вновь выделенную страницу, а в старой странице модифицируется значение размера свободной памяти. Естественно, модифицируются ссылки по списку листовых страниц.
- Чтобы обеспечить доступ от корня дерева к заново заведенной странице, необходимо соответствующим образом модифицировать внутреннюю страницу, являющуюся предком ранее существовавшей листовой страницы, т.е. вставить в нее соответствующее значение ключа и ссылку на новую страницу. При выполнении этого действия может снова произойти переполнение теперь уже внутренней страницы, и она будет расщеплена на две. В результате потребуется вставить значение ключа и ссылку на новую страницу во внутреннюю страницу-предка выше по иерархии и т.д.
- Предельным случаем является переполнение корневой страницы В-дерева. В этом случае она тоже расщепляется на две, и заводится новая корневая страница дерева, т.е. его глубина увеличивается на единицу.

При удалении записи выполняются следующие действия:

- Поиск записи по ключу. Если запись не найдена, то удалять ничего не нужно.
- Реальное удаление записи в буфере, в который прочитана соответствующая листовая страница.

- Если после выполнения этой подоперации размер занятой в буфере области оказывается таковым, что его сумма с размером занятой области в листовых страницах, являющихся левым или правым братом данной страницы, больше, чем размер страницы, операция завершается.
- Иначе производится слияние с правым или левым братом, т.е. в буфере производится новый образ страницы, содержащей общую информацию из данной страницы и ее левого или правого брата. Ставшая ненужной листовая страница заносится в список свободных страниц. Соответствующим образом корректируется список листовых страниц.
- Чтобы устранить возможность доступа от корня к освобожденной странице, нужно удалить соответствующее значение ключа и ссылку на освобожденную страницу из внутренней страницы - ее предка. При этом может возникнуть потребность в слиянии этой страницы с ее левым или правыми братьями и т.д.
- Предельным случаем является полное опустошение корневой страницы дерева, которое возможно после слияния последних двух потомков корня. В этом случае корневая страница освобождается, а глубина дерева уменьшается на единицу.

Как видно, при выполнении операций вставки и удаления свойство сбалансированности В-дерева сохраняется, а внешняя память расходуется достаточно экономно.

Проблемой является то, что при выполнении операций модификации слишком часто могут возникать расщепления и слияния. Чтобы добиться эффективного использования внешней памяти с минимизацией числа расщеплений и слияний, применяются более сложные приемы, в том числе:

- упреждающие расщепления, т.е. расщепления страницы не при ее переполнении, а несколько раньше, когда степень заполненности страницы достигает некоторого уровня;
- переливания, т.е. поддержание равновесного заполнения соседних страниц;
- слияния 3-в-2, т.е. порождение двух листовых страниц на основе содержимого трех соседних.

Следует заметить, что при организации мультидоступа к В-деревьям, характерного при их использовании в СУБД, приходится решать ряд нетривиальных проблем. Конечно, грубые решения очевидны, например монопольный захват В-дерева на все выполнение операции модификации. Но существуют и более тонкие решения.

Сбалансированное дерево автоматически не уравновешивает распределение ключей в пределах дерева так, чтобы половина ключей находилась бы на одной стороне В-дерева, а другая половина — на другой.

Очевидно, что нет необходимости перестраивать дерево всякий раз, когда добавляются или удаляются ключи. Однако если ключи добавляются или удаляются только на одной стороне дерева, то распределение индексных ключей может стать неравномерным, с изрядным числом разреженных и даже опустошенных блоков по одну сторону дерева. В этом случае индекс рекомендуется перестроить.

На В-деревьях для извлечения данных по запросу может использоваться механизм *быстрого полного просмотра* (*fast full scan*). Этот механизм дает существенные преимущества, если все запрошенные из конкретной таблицы данные могут быть получены только из индекса. При быстром полном просмотре эффективный многоблочный ввод/вывод, обычно применяемый для полных просмотров таблиц, используется для прочтения всех листовых блоков В-дерева. Поскольку число листовых блоков индекса, скорее всего, намного меньше, чем блоков данных в таблице, для выполнения запроса требуется просмотреть меньшее число блоков. Поэтому просмотр индекса совершится значительно быстрее, чем полный просмотр таблицы, хотя иногда неравномерное распределение ключей снижает эффективность быстрого полного просмотра, поскольку требуется просмотреть большее число листовых блоков (содержащих малое или вообще нулевое число элементов). При этом следует учитывать наличие или отсутствие в таблице пустых значений, которые, как было сказано выше, в индекс не заносятся.

В-деревья можно использовать для поиска данных, как по условиям равенства, так и по условиям неравенства. Это единственный тип индексов, который можно использовать для предикатов неравенства: LIKE, BETWEEN, “>”, “>=”, “<”, “<=”. Исключение представляет случай использования предиката LIKE при сравнении с шаблоном вида ‘%*выражение*’ или ‘_*выражение*’. В-деревья хранят только непустые значения ключей, так что можно построить разреженное В-дерево.

Хешированные индексы

Хешированные индексы реализованы в Oracle в виде хешированных кластеров. *Хешированный кластер* — специализированный вид организации данных, обеспечивающий быстрый доступ к строкам таблицы. При обращении к хешированному кластеру по значению кластерного ключа применяется *функция хеширования* (*hashing*), результатом которой является значение хешированного ключа и адрес блока данных. Хешированный кластер группирует в одном блоке строки, содержащие одинаковые значения этой функции от ключей. На любой таблице можно построить только один хешированный индекс.

Доступ к таблице посредством В-дерева требует выполнения, по меньшей мере, двух операций ввода/вывода, а обычно больше (если таб-

лица, а потому и дерево ее индекса, большая). Доступ к хешированному кластеру потребует один вызов функции хеширования и одну операцию ввода/вывода для кластера.

Хешированные кластеры целесообразно использовать для больших таблиц, поиск по которым, как правило, осуществляется с условиями равенства по ключевому столбцу (столбцам). При этом значения ключей не модифицируются. Поэтому заранее можно точно определять число значений хешированных ключей и размер кластера. В дополнение к этому, ключевой столбец (столбцы) должен быть высокоселективен.

Главным преимуществом хешированного индекса по сравнению с B-деревом является лучшая производительность выборки, вставки, обновления и удаления записей, если используются условия равенства для всех ключевых столбцов кластера.

Битовые индексы

Битовые индексы обеспечивают быстрое обращение к данным больших таблиц, когда доступ организуется по столбцам с низкой или средней селективностью с использованием различных сочетаний условий равенства. Битовый индекс построен в виде *двоичной карты* (*bitmap*) по значениям ключа. Это означает, что для каждой строки таблицы в двоичной карте, то есть в определенном бите некоторой последовательности байтов, поставлена 1 или 0 (“да” или “нет”) в соответствии со значением ключа конкретной строки. Во время обработки запросов оптимизатор Oracle может динамически преобразовывать элементы индексов битовой карты в идентификаторы строк.

Пример организации битового индекса по низкоселективному столбцу «пол» показан в таблицах.

Таблица

Row ID	ФИО	пол
1	Попов	М
2	Иванова	Ж
3	Сидорова	Ж

Bitmap индекс по столбцу «пол»

Row ID	М	Ж
1	1	0
2	0	1
3	0	1

Битовые карты нецелесообразно применять для часто обновляемых столбцов, поскольку даже простое изменение одной строки обычно приводит к копированию всей соответствующей секции битовой карты. В приведенном примере столбец «пол» вообще не обновляется, так что если данные из таблицы удаляются редко, можно использовать данный индекс без снижения производительности.

Еще необходимо учесть, что Oracle сжимает хранимые битовые карты. В результате для индекса битовой карты может потребоваться дисковое пространство, составляющее 5-10% пространства, необходимого для

обычного индекса. Таким образом, можно применять индексы на основе битовых карт по отношению к любому низкоселективному столбцу, часто используемому в конструкции WHERE, при условии, что набор значений этого столбца ограничен. Битовые индексы можно использовать для поиска только по условиям равенства (“=”, IN). Если необходим доступ по интервалу индексированных значений, то предпочтительнее использовать В-деревья.

Индекс-таблицы

Индекс-таблица – это таблица, которая физически построена в виде двоичного дерева относительно своего первичного ключа. Начиная с Oracle8, существует возможность определить таблицу, которая одновременно является и собственным индексом, что устраняет ведение двух отдельных структур. Как правило, это таблицы с короткими строками, обращение к которым всегда производится или по первичному ключу, или полным сканированием. Данные в таких таблицах отсортированы по значениям столбца первичного ключа и сохраняются так, как если бы вся таблица целиком содержалась в одном индексе.

Чтобы минимизировать объем работы, необходимой для активного управления индексами, следует использовать индекс-таблицу лишь в тех случаях, когда данные очень статичны. Не рекомендуется использовать индекс-таблицы, если строки имеют относительно большую длину, например, свыше 20% от размера блока. В этом случае лучшим выбором является использование обычных таблиц и индексов.

Для создания индекс-таблиц в команде CREATE TABLE указываются ключевые слова ORGANIZATION INDEX.

Пример создания индексно-организованной таблицы:

```
CREATE TABLE TabIndex
(At1 NUMBER(4) PRIMARY KEY,
 At2 VARCHAR(40)
)
 ORGANIZATION INDEX;
```

Существуют некоторые ограничения при работе с индекс-таблицами. Наиболее важно, что их строки не имеют идентификаторов (ROWID), поэтому не могут быть созданы никакие дополнительные индексы, за исключением обязательного первичного ключа.

Следует отметить, что индекс-таблицы активно используются и в других СУБД, например, Microsoft SQL Server по умолчанию создает именно такие таблицы.

Заключение

К сожалению, ограниченный объем учебного пособия не позволил автору включить в него все аспекты технологий баз данных, которые могут потребоваться специалисту в будущей деятельности. Однако, тех фундаментальных базовых знаний, которые можно получить при изучении всех пяти глав пособия, вполне достаточно для дальнейшего самообразования с помощью многочисленных информационных ресурсов, которые прекрасно дополнят полученные знания в области систем баз данных.

Принимая во внимание высокую динамику развития технологий баз данных, можно считать данное пособие всего лишь стартовым учебным материалом, на основе которого будут формироваться новые знания на протяжении всей профессиональной карьеры выпускника технического вуза, работающего в сфере информационных технологий.

Библиографический список

1. Дейт, К. Введение в системы баз данных: пер. с англ. /К.Дж. Дейт. 8-е издание. – М.: Вильямс , 2006. – 1326 с.
2. Ульман, Д. Введение в системы баз данных: пер. с англ. /Д.Ульман, Д.Уидом. – М.: Лори, 2000. – 512 с.
3. Грибер, М. Введение в SQL / М.Грибер, М., Лори, 1996. – 379 с.
4. Базы данных: Учебник для ВУЗов / Под ред.А.Д.Хомоненко — СПб: Корона принт, 2000. – 416 с.
5. Колби, Дж. SQL для начинающих: пер. с англ. / Джон Колби, Пол Уилтон. . – М.: Вильямс, · 2006. – 496 с.
6. Кевин, Кл. SQL: справочник: пер. с англ. / Кл. Кевин. 2-е издание. – М.: Кудиц-Образ, 2006. - 832 с.
7. Полякова, Л. Основы SQL. Курс лекций: учебное пособие / Л.Н. Полякова – М.: ИНТУИТ.РУ, 2004. - 368 с.
8. Абрамсон, Й.. Oracle 10g: Первое знакомство/ Й. Абрамсон, М. Кори, М. Эбби. – М.: Лори , 2007. – 348 с.
9. Базы данных. Рабочая программа, методические указания к лабораторным работам и курсовому проектированию, варианты заданий. / сост. С.Ю.Ржеуцкая, М.Н.Артюгин — Вологда: ВоГТУ, 2007. – 48 с.

Учебное издание

Светлана Юрьевна Ржеуцкая

БАЗЫ ДАННЫХ
ЯЗЫК SQL

Учебное пособие

Редактор – И.Т. Куликова

Подписано в печать 31.08.2010. Формат 60 × 90/$_{16}$
Бумага писчая. Печать офсетная.
Усл.-п.л. 9,9. Тираж экз. Заказ №

Отпечатано: РИО, ВоГТУ 160000, г. Вологда, ул. Ленина, 15